# Introduction to Soft Computing

Eva Volna

bookboon
LEARNING

Eva Volná

# Introduction to Soft Computing

Introduction to Soft Computing
1st edition

ISBN 978-87-403-0573-9

# Contents

# Abstract

The book is structured conceptually to include four main parts physically organised into chapters.

*Chapter 1* is an introduction to soft computing providing a brief history about its origins and what it may cover. It was concluded that an exact notion of soft computing has not been defined yet as it is still an area newly shaping up. Therefore, an introduction to the basic soft computing as proposed by its inventor, Dr. Lotfi Zadeh, is given.

The other fields that constitute the area of soft computing are fuzzy computing, evolutionary computing, artificial neural networks, and probabilistic computing.

*Chapter 2* starts with an introduction to fuzzy sets and fuzzy control applications.

*Chapter 3* covers the components of evolutionary computing with the most common evolutionary computing technique, namely genetic algorithms, genetic programming and differential evolution.

*Chapter 4* is an introduction to the field of artificial neural networks. The basic models are described here, e.g. the perceptron, the multilayer network, the Kohonen self-organizing map and the Hopfield networks.

C*hapter 5* introduces the probabilistic computing in short.

The cooperation between some Soft Computing fields is demonstrated in the *Conclusion*

**doc. RNDr. PaedDr. Eva Volná, PhD.**
Dept. of Computer Science
University of Ostrava
30th dubna st. 22
701 03 Ostrava
CZECH REPUBLIC

Phone: +420 597 092 184
Fax: +420 596 120 478
E-mail: eva.volna@osu.cz

# 1 What does Soft Computing mean?

## 1.1 Definition of Soft Computing

Prior to 1994 when Zadeh (Zadeh 1994) first defined "soft computing", the currently-handled concepts used to be referred to in an isolated way, whereby each was spoken of individually with an indication of the use of fuzzy methodologies. Although the idea of establishing the area of soft computing dates back to 1990 (Zadeh 2001), it was in (Zadeh 1994) that Zadeh established the definition of soft computing in the following terms:

"*Basically, soft computing is not a homogeneous body of concepts and techniques. Rather, it is a partnership of distinct methods that in one way or another conform to its guiding principle. At this juncture, the dominant aim of soft computing is to exploit the tolerance for imprecision and uncertainty to achieve tractability, robustness and low solutions cost. The principal constituents of soft computing are fuzzy logic, neurocomputing, and probabilistic reasoning, with the latter subsuming genetic algorithms, belief networks, chaotic systems, and parts of learning theory. In the partnership of fuzzy logic, neurocomputing, and probabilistic reasoning, fuzzy logic is mainly concerned with imprecision and approximate reasoning; neurocomputing with learning and curve-fitting; and probabilistic reasoning with uncertainty and belief propagation*".

**Figure 1:** Prof. Lotfi. A. Zadeh.
(adapted from http://www-bisc.eecs.berkeley.edu)

There have been various subsequent attempts to further hone this definition, with differing results, and among the possible alternative definitions, perhaps the most suitable is the one presented in (Li & Ruan & van der Wal 1998):

"*Every computing process that purposely includes imprecision into the calculation on one or more levels and allows this imprecision either to change (decrease) the granularity of the problem, or to "soften" the goal of optimization at some stage, is defined as to belonging to the field of soft computing*".

Soft computing could therefore be seen as a series of techniques and methods so that real practical situations could be dealt with in the same way as humans deal with them, i.e. on the basis of intelligence, common sense, consideration of analogies, approaches, etc. In this sense, soft computing is a family of problem-resolution methods headed by approximate reasoning and functional and optimisation approximation methods, including search methods. Soft computing is therefore the theoretical basis for the area of intelligent systems and it is evident that the difference between the area of artificial intelligence and that of intelligent systems is that the first is based on hard computing and the second on soft computing. Soft Computing is still growing and developing.

From this other viewpoint on a second level, soft computing can be then expanded into other components which contribute to a definition by extension, such as the one first given. From the beginning (Bonissone 2002), the components considered to be the most important in this second level are probabilistic reasoning, fuzzy logic and fuzzy sets, neural networks, and genetic algorithms, which because of their interdisciplinary, applications and results immediately stood out over other methodologies such as the previously mentioned chaos theory, evidence theory, etc. The popularity of genetic algorithms, together with their proven efficiency in a wide variety of areas and applications, their attempt to imitate natural creatures (e.g. plants, animals, humans) which are clearly soft (i.e. flexible, adaptable, creative, intelligent, etc.), and especially the extensions and different versions, transform this fourth second-level ingredient into the well-known evolutionary algorithms which consequently comprise the fourth fundamental component of soft computing, as shown in the following diagram, see Figure 2.

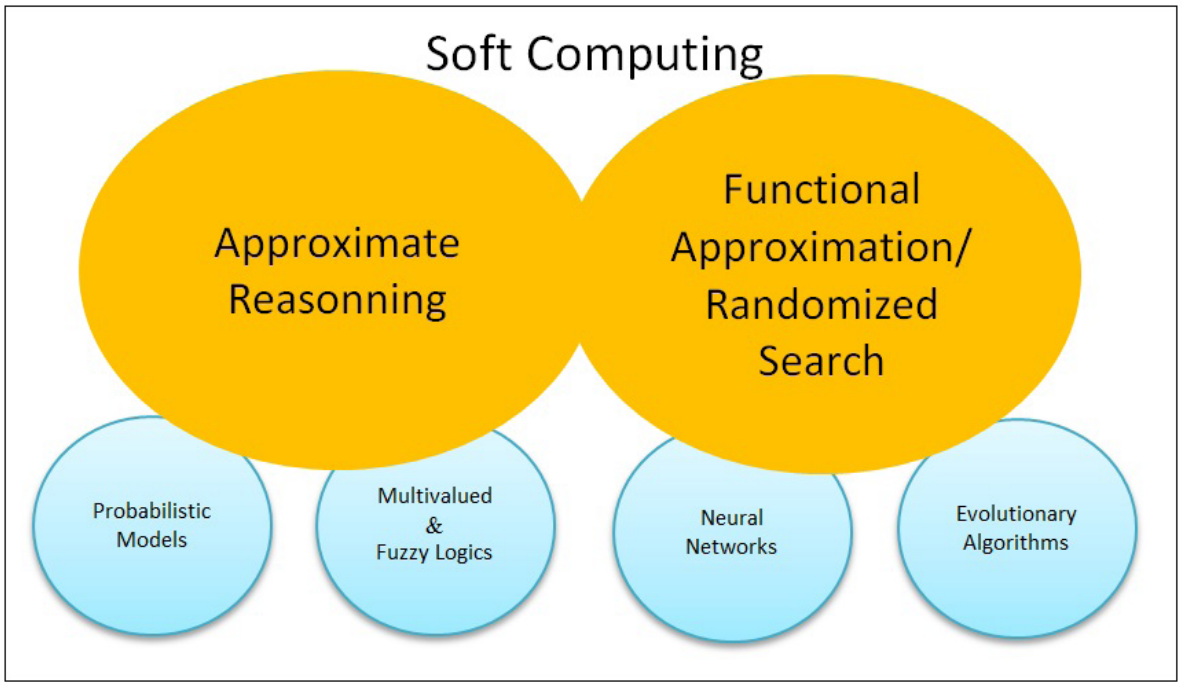



**Figure 2:** What does Soft Computing mean? (adapted from http://modo.ugr.es)

## 1.2 Conception of Soft Computing

From this last conception of soft computing playing fuzzy sets and fuzzy logic a necessarily basic role, we can describe the following other areas emerging around it simply by considering some of the possible combinations which can arise:

1. From the first level and beginning with approximate reasoning methods, when we only concentrate on probabilistic models, we encounter the Dempster-Shafer theory and Bayesian networks. However, when we consider probabilistic methods combined with fuzzy logic, and even with some other multi-valued logics, we encounter what we could call hybrid probabilistic models, fundamentally probability theory models for fuzzy events, fuzzy event belief models, and fuzzy influence diagrams.
2. When we look at the developments directly associated with fuzzy logic, fuzzy systems and in particular fuzzy controllers stand out. Then, arising from the combination of fuzzy logic with neural networks and EA are fuzzy logic-based hybrid systems, the foremost exponents of which are fuzzy neural systems, controllers adjusted by neural networks (neural fuzzy systems which differ from the previously mentioned fuzzy neural systems), and fuzzy logic-based controllers which are created and adjusted with evolutionary algorithms.
3. Moving through the first level to the other large area covered by soft computing (functional approach/optimization methods) the first component which appears is that of neural networks and their different models. Arising from the interaction with fuzzy logic methodologies and EA methodologies are hybrid neural systems, and in particular fuzzy control of network parameters, and the formal generation and weight generation in neural networks.
4. The fourth typical component of soft computing and perhaps the newest yet possibly most up-to-date is that of evolutionary algorithms, and associated with these are four large, important areas: evolutionary strategies, evolutionary programming, genetic algorithms, and genetic programming. If we were only to focus on these last areas, we could consider that in this case the amalgam of methodologies and techniques associated with soft computing culminate in three important lines: fuzzy genetic systems, bio inspired systems, and applications for the fuzzy control of evolutionary parameters.

## 1.3 Importance of Soft Computing

The aim of Soft Computing is to exploit tolerance for imprecision, uncertainty, approximate reasoning, and partial truth in order to achieve close resemblance with human-like decision making. Soft Computing is a new multidisciplinary field, to construct a new generation of Artificial Intelligence, known as *Computational Intelligence*.

The main goal of Soft Computing is to develop intelligent machines and to solve nonlinear and mathematically unmodelled system problems (Zadeh 1994) and (Zadeh 2001). The applications of Soft Computing have proved two main advantages. *First*, it made solving nonlinear problems, in which mathematical models are not available, possible. *Second*, it introduced the human knowledge such as cognition, recognition, understanding, learning, and others into the fields of computing. This resulted in the possibility of constructing intelligent systems such as autonomous self-tuning systems, and automated designed systems.

As stated in (Verdegay 2003), since the fuzzy boom of the 1990s, methodologies based on fuzzy sets (i.e. soft computing) have become a permanent part of all areas of research, development and innovation, and their application has been extended to all areas of our daily life: health, banking, home, and are also the object of study on different educational levels. Similarly, there is no doubt that thanks to the technological potential that we currently have, computers can handle problems of tremendous complexity (both in comprehension and dimension) in a wide variety of new fields.

As we mentioned above, since the 1990s, evolutionary algorithms have proved to be extremely valuable for finding good solutions to specific problems in these fields, and thanks to their scientific attractiveness, the diversity of their applications and the considerable efficiency of their solutions in intelligent systems, they have been incorporated into the second level of soft computing components.

Evolutionary algorithms, however, are merely another class of heuristics, or metaheuristics, in the same way as Tabu Search, Simulated Annealing, Hill Climbing, Variable Neighbourhood Search, Estimation Distribution Algorithms, Scatter Search, Reactive Search and very many others are. Generally speaking, all these heuristic algorithms (metaheuristics) usually provide solutions which are not ideal, but which largely satisfy the decision-maker or the user. When these act on the basis that satisfaction is better than optimization, they perfectly illustrate Zadeh's famous sentence (Zadeh 1994):

"*…in contrast to traditional hard computing, soft computing exploits the tolerance for imprecision, uncertainty, and partial truth to achieve tractability, robustness, low solution-cost, and better rapport with reality*".

## 1.4 The Soft Computing – development history

The following two schemes[1] show development history of Soft Computing in brief.

| SC | = | EC | + | NN | + | FL |
|---|---|---|---|---|---|---|
| *Soft Computing* | | *Evolutionary Computing* | | *Neural Network* | | *Fuzzy Logic* |
| **Zadeh 1981** | | **Rechenberg 1960** | | **McCulloch 1943** | | **Zadeh 1965** |

| EC | = | GP | + | ES | + | EP | + | GA |
|---|---|---|---|---|---|---|---|---|
| *Evolutionary Computing* | | *Genetic Programming* | | *Evolution Strategies* | | *Evolutionary Programming* | | *Genetic Algorithms* |
| **Rechenberg 1960** | | **Koza 1992** | | **Rechenberg 1965** | | **Fogel 1962** | | **Holland 1970** |

# 2 Fuzzy Computing

In the real word there exists much fuzzy knowledge, that is, knowledge which is vague, imprecise, uncertain, ambiguous, inexact, or probabilistic in nature.

Human can use such information because the human thinking and reasoning frequently involve fuzzy information, possibly originating from inherently inexact human concepts and matching of similar rather than identical experience.

The computing system, based upon classical set theory and two-valued logic, cannot give answers to some questions as a human does, because they do not have completely true answers.

We want the computing systems not only to give human-like answers but also to describe their reality levels. These levels need to be calculated using imprecision and the uncertainty of facts as well as rules that were applied.

## 2.1 Fuzzy sets

Fuzzy Logic is built on *The Fuzzy Set Theory* which was introduced to the world by Lotfi Zadeh in 1965 for the first time. The invention, or proposition, of *Fuzzy Sets* was motivated by the need to capture and represent the real world with its fuzzy data due to uncertainty. Uncertainty can be caused by imprecision in measurement due to imprecision of tools or other factors. Uncertainty can also be caused by vagueness in the language objects and situations. Lotfi Zadeh realized that the *Crisp Set Theory* is not capable of representing those descriptions and classifications in many cases. In fact, *Crisp Sets* do not provide adequate representation. We use linguistic variables often to describe, and maybe classify, physical objects and situations.

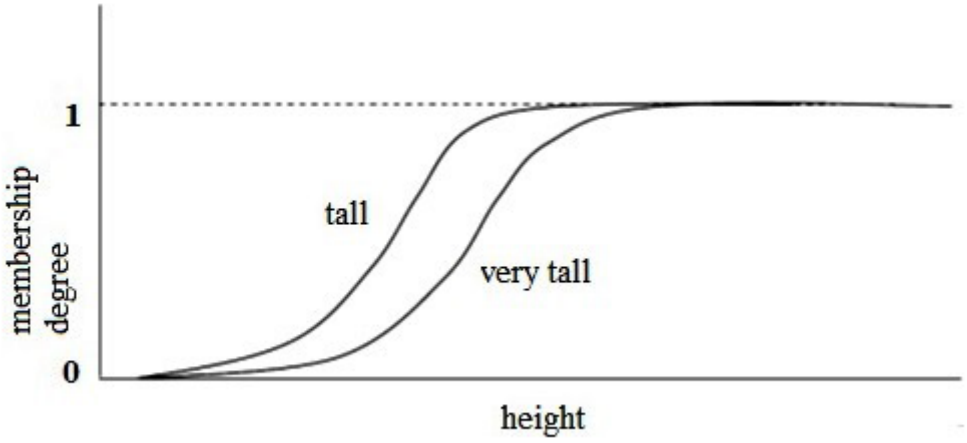



**Figure 3:** Fuzzy set representation.

Instead of avoiding or ignoring uncertainty, Lotfi Zadeh developed a set theory that captures this uncertainty. The goal was to develop a set theory and a resulting logic system that are capable of coping with the real world. Therefore, rather than defining Crisp Sets, where elements are either in or out of the set with the absolute certainty, Zadeh proposed the concept of a *Membership Function*. An element can be in the set with a degree of membership and out of the set with a degree of membership. Figure 3 illustrates the use of Fuzzy Sets to represent the notion of a tall person. It also shows how we can differentiate between the notions of tall and very tall, resulting in a more accurate model than the classical set theory.

### 2.1.1 Basic properties of fuzzy sets

Fuzzy set theory is primarily concerned with quantifying and reasoning using natural language in which many words have ambiguous meanings. It can also be thought of as an extension of the traditional crisp set, in which each element must either be in or not be in a set. Formally, the process by which individuals from a universal set $X$ are determined to be either members or non-members of a crisp set can be defined by a *characteristic* or *discrimination function*. For a given crisp set $A$ this function assigns a value $\mu_A(x)$ to every $x \in X$ such that

$$\mu_A(x) = \begin{cases} 1 & f \quad x \in A \\ 0 & f \quad x \notin A \end{cases}$$

Thus, the function maps elements of the universal set to the set containing 0 and 1 (Figure 4). This can be indicated by

$$\mu_A : \mathrm{X} \rightarrow \{0, 1\}$$

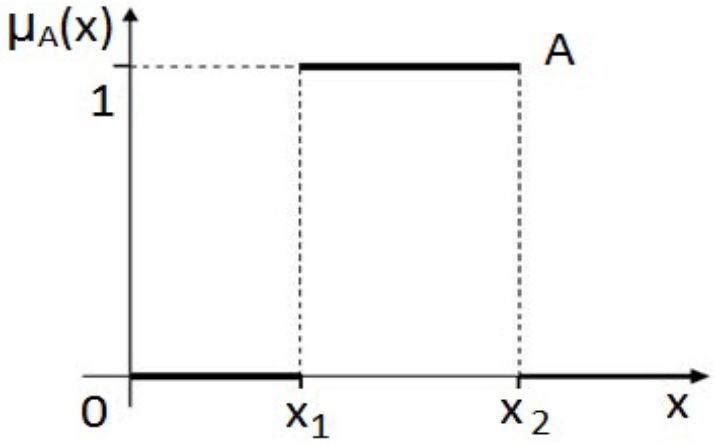



**Figure 4:** Graphical representation of crisp sets

This kind of function can be generalized such that the values assigned to the elements of the universal set fall within a specified range and are referred to as the membership grades of these elements in the set. Larger values denote higher degrees of the set membership. Such function is called a membership function $\mu_A$ by which a fuzzy set $A$ is usually defined and represents the degree of membership of $x$ in $A$. This function can be indicated by

$$\mu_A : X \rightarrow [0, 1],$$

where X refers to the universal set defined in a specific problem, and [0,1] denotes the interval of real numbers from 0 to 1, inclusively.

In the case of Crisp Sets, the members of a set are either out of *the s*et, with the membership degree of zero, or in the set, with the value one being the degree of membership. Therefore, Crisp Sets $\subseteq$ Fuzzy Sets or in other words, Crisp Sets are Special cases of Fuzzy Sets.

There are four ways of representing fuzzy membership functions, namely, *graphical representation*, *tabular and list representation*, *geometric representation*, and *analytic representation*. Graphical representation is the most common in the literature. Figure 3 above is an example of the graphical representation of fuzzy membership functions. Tabular and list representations are used for finite sets. In this type of representation, each element of the set is paired with its degree of membership. Two different notations have been used in the literature for tabular and list representation. The following example illustrates the two notations for the same membership function.

$$\mu_A = \{ <x_1, 0.8>, <x_2, 0.3>, <x_3, 0.5>, <x_4, 0.9> \}$$

$$\mu_A = 0.8/x_1 + 0.3/x_2 + 0.5/x_3 + 0.9/x_4$$

The third method of representation is the geometric representation and is also used for representing finite sets. For a set that contains *n* elements, *n*-dimentional Euclidean space is formed and each element may be represented as a coordinate in that space. Finally analytical representation is another alternative to graphical representation in representing infinite sets, e.g., a set of real numbers. The following example illustrates both graphical and analytical representation of the same fuzzy function:

$$\mu(A) = \begin{cases} x-5 & \text{when } 5 \le x \le 6 \\ 7-x & \text{when } 6 \le x \le 7 \\ 0 & \text{otherwise} \end{cases}$$

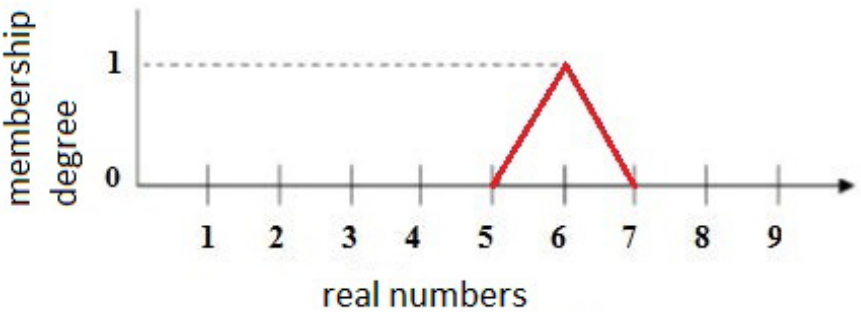



**Figure 5:** Graphical representation of the analytical representation given above

With the additional restriction that the membership function must capture an intuitive conception of a set of real numbers surrounding a given central real number, or interval of real numbers. In this context, the example above illustrates the concept of the fuzzy number "about six", "around six", or "approximately six".

Another very important property of fuzzy sets is the concept of α – *cut* (alpha cut). ∝-cuts reduce a fuzzy set into an extracted crisp set. The value α represents a membership degree, i.e. α ∈ [0, 1]. The α-cut of a fuzzy set *A* is the crisp set (*A* – α), i.e. the set of all elements whose membership degrees in *A* are ≥ α (Kohout 1999).

### 2.1.2 Basic properties of fuzzy sets

The basic operations on fuzzy sets (Figure 6) are Fuzzy Complement (Figure 7), Fuzzy Union (Figure 8), and Fuzzy Intersection (Figure 9). These operations are defined as follows:

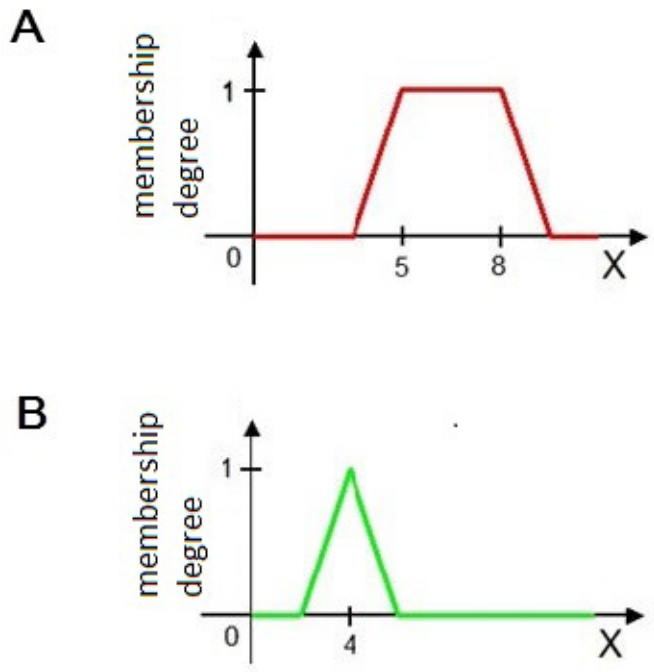



**Figure 6:** Fuzzy sets A and B

**Fuzzy Complement** of *A*: $\overline{A}(x): \mu_{\overline{A}} = 1 - \mu_A(x)$ for all $x \in X$

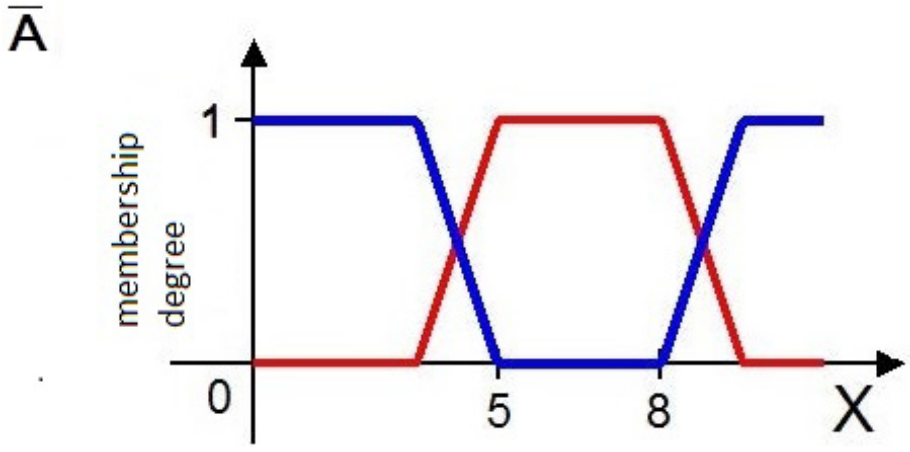



**Figure 7:** Fuzzy Complement

**Fuzzy Union** of $A$ and $B$: $(A \cup B)$: $\mu_{A \cup B} = \max\{\mu_A(x); \mu_B(x)\}$ for all $x \in X$
Notice that $A \cup \overline{A} \neq X$, which violates the *Law of Excluded Middle*.

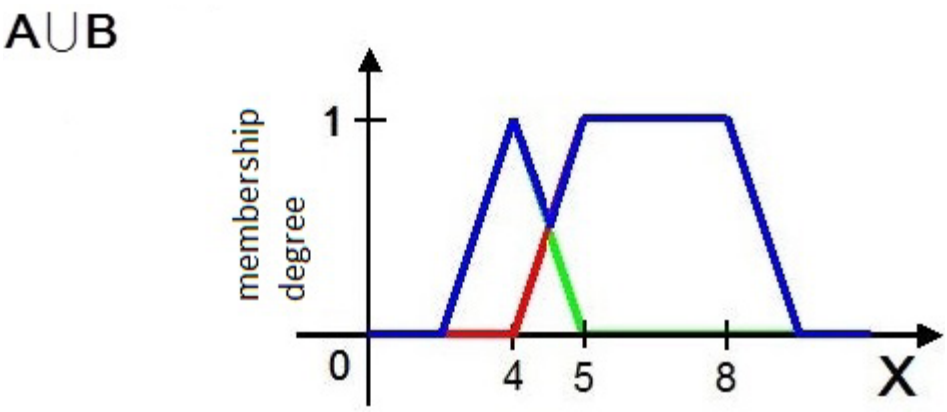



**Figure 8:** Fuzzy Union

**Fuzzy Intersection** of $A$ and $B$: $(A \cap B)$: $\mu_{A \cap B} = \min\{\mu_A(x); \mu_B(x)\}$ for all $x \in X$
Notice that $A \cap \overline{A} \neq \phi$, which violates the Law of Contradiction.

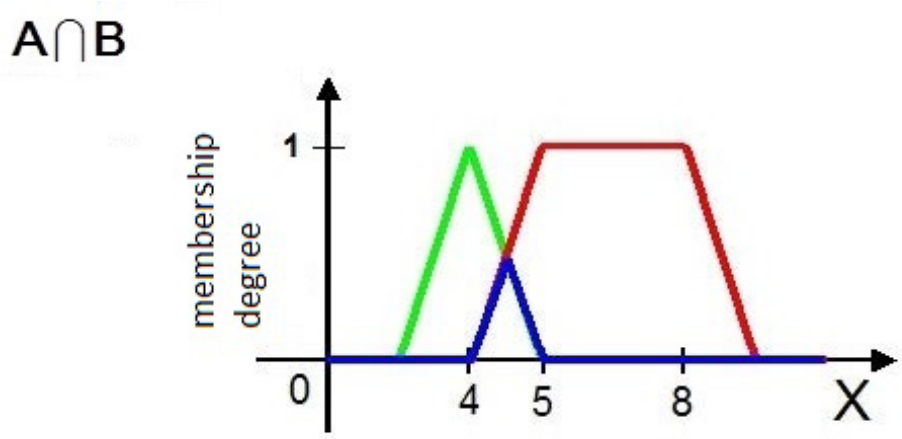



**Figure 9:** Fuzzy Intersection

### 2.1.3 Fuzzy arithmetic

Fuzzy Arithmetic uses arithmetic on closed intervals. The basic fuzzy arithmetic operations are defined as follows:

**Addition:** [a, b] + [c, d] = [a+ c, b + d]

**Subtraction:** [a, b] – [c, d] = [a – d, b – c]

**Multiplication:** [a,b].[c,d] = [min(ac, ad, bc, bd), max(ac, ad, bc, bd)]

**Division:** [a, b] / [c, d] = [a, b] . [1/d, 1/c] = [min(a/c, a/d, b/c, b/d),max(a/c, a/d, b/c, b/d)]

Figure 10 illustrates graphical representations of fuzzy addition and subtraction.

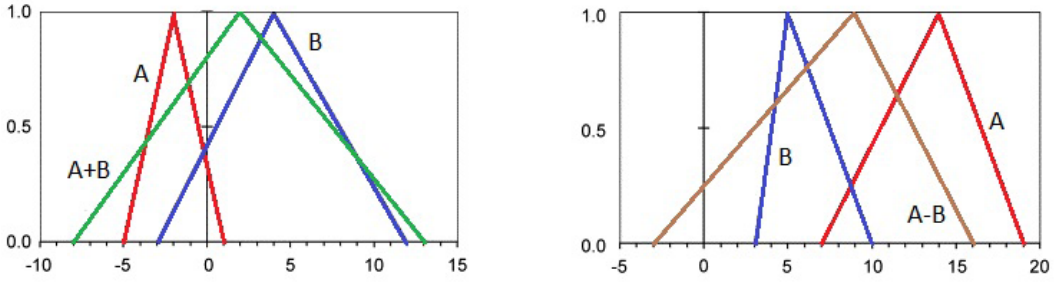



**Figure 10:** Example of fuzzy addition and subtraction

### 2.1.4 Fuzzy relations

**Properties of fuzzy relations:**

Fuzzy Relations were introduced to supersede classical crisp relations. Rather than just describing the full presence or full absence of association of elements of various sets in the case of crisp relations, Fuzzy Relations describe the degree of such association. This gives fuzzy relations the capability to capture the uncertainty and vagueness in relations between sets and elements of a set. Furthermore, it enables fuzzy relations to capture the broader concepts expressed in fuzzy linguistic terms when describing the relation between two or more sets. For example, when classical sets are used to describe the equality relation, it can only describe the concept "*x* is equal to *y*" with absolute certainty, i.e., if x is equal to *y* with unlimited precision, then *x* is related to *y*, otherwise *x* is not related to *y*, even if it was slightly different. Thus, it is not possible to describe the concept "*x* is approximately equal to *y*". Fuzzy Relations make the description of such a concept possible. Table 1 provides comparison of the special properties of Crisp and Fuzzy relations, $E_x$ is the Equality Relation and $O_x$ is the Empty Relation (Bandler and Kohout 1988). It is important to note here that the concept of local reflexivity was introduced for the first time in Crisp Relational Theory by Bandler and Kohout in 1977. The fast fuzzy relational algorithms that employ local reflexivity in fuzzy computing were introduced also by Bandler and Kohout in 1982.

| Property | Crisp | Fuzzy |
|---|---|---|
| Covering | $\Leftrightarrow \forall i \in J, \exists j \in J \mid R_{ij} = 1$ | $\Leftrightarrow \forall i \in J, \exists j \in J \mid R_{ij} = 1$ |
| Locally reflexive | $\Leftrightarrow \forall i \in J, R_{ii} = \bigvee_j (R_{ij} \vee R_{ji})$ | $\Leftrightarrow \forall i \in J, R_{ii} = \bigvee_j (R_{ij} \vee R_{ji})$ |
| Reflexive | $\Leftrightarrow$ Covering and locally reflexive | $\Leftrightarrow$ Covering and locally reflexive |
| Transitive | $\Leftrightarrow R^2 \subseteq R$ | $\Leftrightarrow R^2 \subseteq R$ |
| Symmetric | $\Leftrightarrow R^T \subseteq R$ | $\Leftrightarrow R^T \subseteq R$ |
| Antisymmetric | $\Leftrightarrow R \cap R^T \subseteq E_x$ | $\Leftrightarrow R_{ij} \wedge R_{ji} = 0$ if $j \neq i$ |
| Strictly Antisymmetric | $\Leftrightarrow R \cap R^T = O_x$ | $\Leftrightarrow \forall i, j \in J, R_{ij} \wedge R_{ji} = 0$ |

**Table 1:** Properties of Crisp vs. Fuzzy Relations

**Representation of Fuzzy Relations**

The most common methods of representing fuzzy relations are lists of $n$-tuples, formulas, matrices, mappings, and directed graphs. A list of $n$-tuples, i.e., ordered pairs, can be used to represent finite fuzzy relations. The tuple consists of a Cartesian product with its membership degree. When the membership has degree zero, the tuple is usually omitted. Suitable formulas are usually used to define infinite fuzzy relations, which involve $n$-dimensional Euclidean space, with $n \geq 2$. Matrices, or $n$-dimensional arrays, are the most common method to represent fuzzy relations. In this method, the entries of the matrix are the membership degrees associated with the $n$-tuple of the Cartesian product. The mapping of fuzzy relations is an extension of the mapping method of classical binary relations. For fuzzy relations, the connections of the mapping diagram are labelled with the membership degree. The same technique is used to extend the directed graph representation of classical relations to represent fuzzy relations.

**Operations on fuzzy relations:**

All the mathematical and logical operations on fuzzy sets explained above are also applicable to fuzzy relations. In addition, there are operations on fuzzy binary relations that do not apply to general fuzzy sets. Those operations are the inverse, the composition, and the BK-products of fuzzy relations.

The inverse of a fuzzy binary relation $R$ on two sets $X$ and $Y$ is also a relation denoted by $R^{-1}$ such that $xR^{-1}y = yRx$. Therefore, for any fuzzy binary relation, $(R^{-1})^{-1} = R$. When using matrix representation, the inverse can be obtained by generating the transpose of the original matrix, i.e., swapping the columns and the rows of the matrix as in the following example.

$$R = \begin{bmatrix} 0.5 & 1 & 0 \\ 1 & 0.8 & 0.2 \\ 0.7 & 0 & 0.3 \end{bmatrix} \qquad R^{-1} = \begin{bmatrix} 0.5 & 1 & 0.7 \\ 1 & 0.8 & 0 \\ 0 & 0.2 & 0.3 \end{bmatrix}$$

**The composition of two fuzzy relations is defined as follows:**

Let $P$ be a fuzzy relation from $X$ to $Y$ and $Q$ be a fuzzy relation from $Y$ to $Z$ such that the membership degree is defined by $P$(x, y) and $Q$(y, z). Then, a third fuzzy relation $R$ from $X$ to $Z$ can be produced by the composition of $P$ and $Q$, which is denoted as $(P \cdot Q)$. Fuzzy relation $R$ is computed by the formula (Klir and Yuan 1995):

$$R(x,z) = (P \cdot Q)(x,z) = \max_{y \in Y} \{\min[P(x,y), Q(y,z)]\}$$

The idea of producing fuzzy relational composition was expanded by Bandler and Kohout in 1977 when they introduced, for the first time, special relational compositions called the Triangle and Square products (Bandler and Kohout 1987). The Triangle and Square products were named after their inventors and became known as BK-products. The BK-products of fuzzy relations proved to be very powerful not only as a mathematical tool for operations on fuzzy sets and fuzzy relations but also as a computational framework for fuzzy logic and fuzzy control. In addition to the set based definitions presented above, many valued logic operations are also implied and are defined as follows:

$$\text{Circle product } (R \circ S)_{ik} = \vee_j (R_{ij} \wedge S_{ik})$$

$$\text{Triangle sub-product } (R \triangleleft S)_{ik} = \wedge_j (R \rightarrow S_{ik})$$

$$\text{Triangle super-product } (R \triangleright S)_{ik} = \wedge_j (R \leftarrow S_{ik})$$

$$\text{Square product } (R - S)_{ik} = \wedge_j (R \equiv S_{ik})$$

Where $R_{ij}$ and $S_{ik}$ represent the fuzzy degree of truth of the propositions $x_i R\ y_j$ and $y_j S\ z_k$, respectively (Kohout 2000).

BK-products have been applied, as a powerful computational tool, in many fields such as computer protection, artificial intelligence, medicine, information retrieval, handwriting classification, urban studies, investment, control, and most recently in quality of service and distributed networking (Moussa and Kohout 2001).

### 2.1.5 Fuzzy rules

*Fuzzy rules* are linguistic IF-THEN- constructions that have the general form "IF *A* THEN *B*" where *A* and *B* are (collections of) propositions containing linguistic variables. *A* is called the *premise* and *B* is the *consequence* of the rule. In effect, the use of linguistic variables and fuzzy IF-THEN- rules exploits the tolerance for imprecision and uncertainty. In this respect, fuzzy logic mimics the crucial ability of the human mind to summarize data and focus on decision-relevant information.

In a more explicit form, if there are *i* rules, each with *k* premises in a system, the *i-th* rule has the following form:

$$If\ a_1\ is\ A_{i,1}\ \Theta\ a_2\ is\ A_{i,2}\ \Theta\ \ldots\ \Theta\ a_k\ is\ A_{i,k}\ \text{then}\ B_i$$

In the above equation *a* represents the crisp inputs to the rule and *A* and *B* are linguistic variables. The operator Θ can be AND or OR or XOR.

*Example:*

If a HIGH flood is expected and the reservoir level is MEDIUM, then water release is HIGH.

Several rules constitute a *fuzzy rule-based system*.

**Mamdani method**

The most commonly used fuzzy inference technique is the so-called Mamdani method. Professor Ebrahim Mamdani of London University built one of the first fuzzy systems to control a steam engine and boiler combination. He applied a set of fuzzy rules supplied by experienced human operators in 1975.

Mamdami method runs in 4 steps:

1. Fuzzification of the input variables.
2. Rule evaluation.
3. Aggregation of the rule outputs.
4. Defuzzification.

*Example:*

Rule: 1

IF *funding* is *adequate* OR *staffing* is *small* THEN *risk* is *low*

Rule: 2

IF *funding* is *marginal* AND *staffing* is *large* THEN *risk* is *normal*

Rule: 3

IF *funding* is *inadequate* THEN *risk* is *high*

Note that *funding*, *staffing* and *risk* are linguistic variables, *inadequate*, *marginal* and *adequate* are linguistic values determined by fuzzy sets on the universe of discourses. Also, *small* and *large* are linguistic values determined by fuzzy sets.

STEP 1: Fuzzification

The first step is to take the crisp inputs, (let funding and staffing be $x_1$ and $y_1$), and determine the degree to which these inputs belong to each of the appropriate fuzzy sets. The crisp input is a numerical input. For instance, let the expert determine a figure between 0–100 to represent funding and staffing, say 35% and 60%.

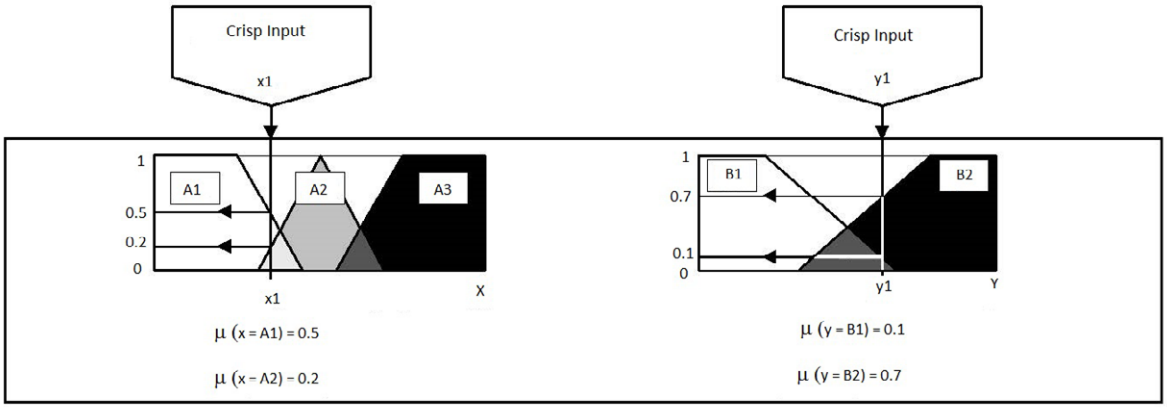



**Figure 11:** The first step is to take the crisp inputs (adapted from http://www.4c.ucc.ie).

STEP 2: Rule evaluation

The second step is to take the fuzzified inputs, $\mu_{(x=A1)} = 0.5$; $\mu_{(x=A2)} = 0.2$; $\mu_{(y=B1)} = 0.1$ and $\mu_{(y=B2)} = 0.7$, and apply them to the antecedents of the fuzzy rules. If a given fuzzy rule has multiple antecedents, the fuzzy operator (**AND** or **OR**) is used to obtain a single number that represents the result of the antecedent evaluation. This number (the truth value) is then applied to the consequent membership function. To evaluate the disjunction of the rule antecedents, we use the **OR** fuzzy operation. Typically, using the fuzzy operation **union**:

$$(A \cup B): \mu_{A \cup B} = \max\{\mu_A(x); \mu_B(x)\}$$

Similarly, in order to evaluate the conjunction of the rule antecedents, we apply the **AND** fuzzy operation **intersection**:

$$(A \cap B): \mu_{A \cap B} = \min\{\mu_A(x); \mu_B(x)\}$$

Mamdami rule evaluation runs in the following steps (Figure 12):

- The result of the antecedent evaluation can be now applied to the membership function of the consequent.
- **Clipping** is a common method of correlating the rule consequent with the truth value of the rule antecedent is to cut the consequent membership function at the level of the antecedent truth. (Figure 13).
- **Scaling** is a better approach for preserving the shape of the fuzzy set. The original membership function of the rule consequent is adjusted by multiplying its membership degrees by the truth value of the rule antecedent. (Figure 13).

Since the top of the membership function is sliced, the clipped fuzzy set loses some information. However, clipping is still often preferred because it involves less complex and faster mathematics, and generates an aggregated output surface that is easier to defuzzify (in Step 4).

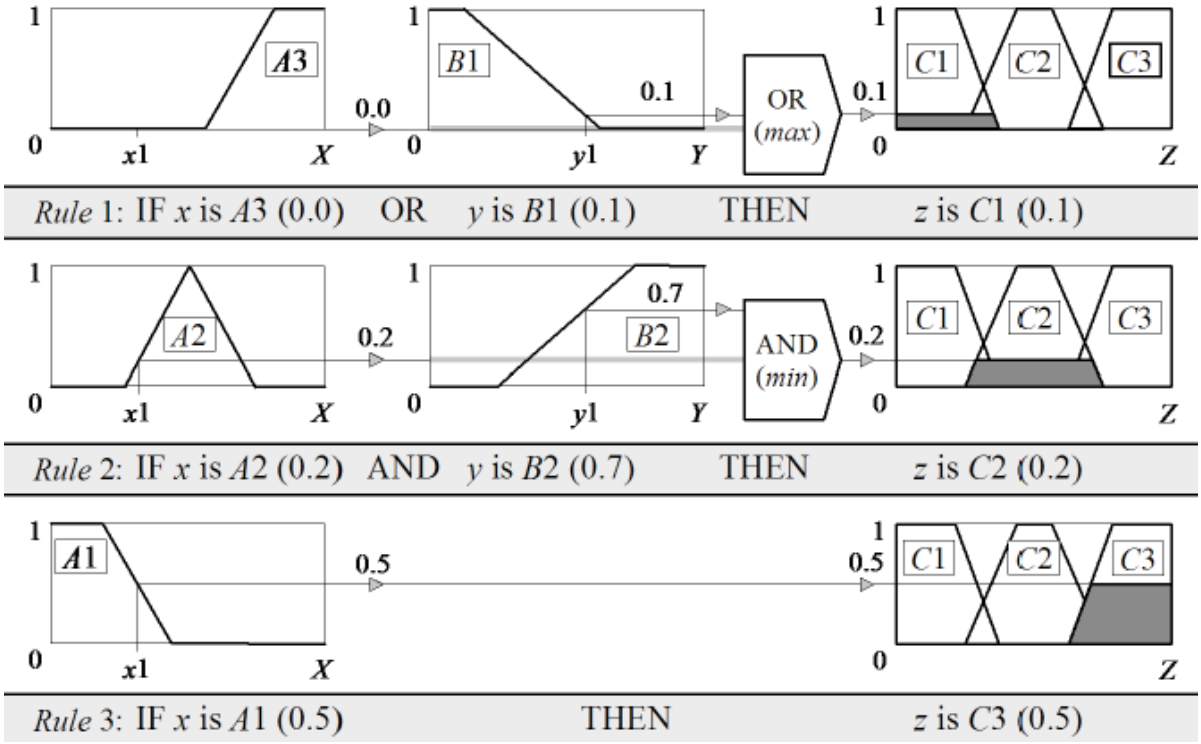



**Figure 12:** Mamdami rule evaluation (adapted from http://www.4c.ucc.ie).

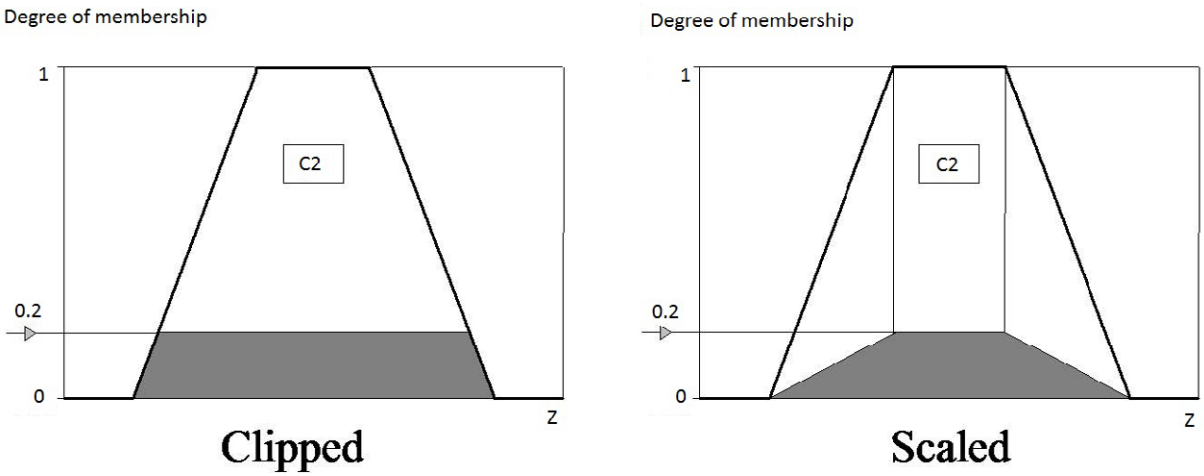



**Figure 13:** Clipping and scalling (adapted from http://www.4c.ucc.ie).

STEP 3: Aggregation of rule outputs (Figure 14)

Aggregation is the process of unification of the outputs of all rules. We take the membership functions of all rule consequents previously clipped or scaled and combine them into a single fuzzy set. The input of the aggregation process is the list of clipped or scaled consequent membership functions, and the output is one fuzzy set for each output variable.

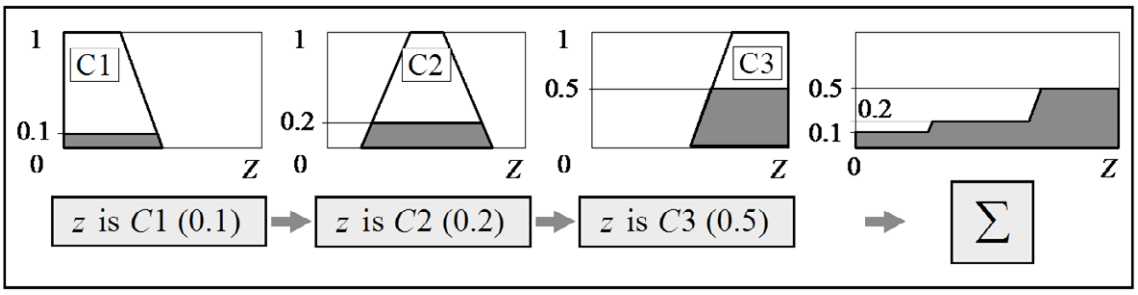



**Figure 14:** Aggregation of rule outputs (adapted from http://www.4c.ucc.ie).

STEP 4: Defuzzification

The last step in the fuzzy inference process is the **defuzzification**. Fuzziness helps us to evaluate the rules, but the final output of a fuzzy system has to be a crisp number. The input for the defuzzification process is the aggregate output fuzzy set and the output is a single number.

There are several defuzzification methods, but probably the most popular one is the **centroid** technique. It finds the point where a vertical line would slice the aggregate set into two equal masses. Mathematically this **centre of gravity (COG)** can be expressed as (Figure 15):

$$COG = \frac{\int_a^b \mu_A(x)x\,dx}{\int_a^b \mu_A(x)\,dx}$$

- The centroid defuzzification method finds a point representing the centre of gravity of the fuzzy set, *A*, on the interval [*a*, *b*].
- A reasonable estimate can be obtained by calculating it over a sample of points.

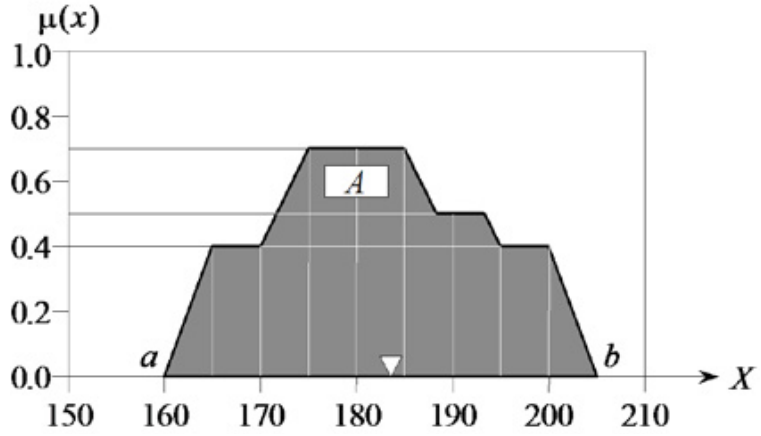



**Figure 15:** Centre of gravity (COG) (adapted from http://www.4c.ucc.ie).

**Sugeno fuzzy interference**

The Mamdani-style inference, as we have just seen, *requires* us to find the centroid of a two-dimensional shape by integrating across a continuously varying function. In general, this process is not computationally efficient. Michio Sugeno suggested using a single spike, **a fuzzy singleton**, as the membership function of the rule consequent. This is a fuzzy set with a membership function that is unity at a single particular point on the universe of discourse and zero everywhere else. Sugeno-style fuzzy inference is very similar to the Mamdani method. Sugeno changed only a rule consequent. He used a mathematical function of the input variable (instead of a fuzzy set).

Format of Sugeno-style fuzzy rule is the following. **IF** $x$ is $A$ **AND** y is $B$ **THEN** $z$ is $f$ ($x$, $y$), where $x$, $y$ and $z$ are linguistic variables; $A$ and $B$ are fuzzy sets on universe of discourses $X$ and $Y$, respectively; and $f$ ($x$, $y$) is a mathematical function. The most commonly used zero-order Sugeno fuzzy model applies fuzzy rules whose **THEN** part takes the following form: **THEN** $z$ is $k$, where $k$ is a constant. In this case, the output of each fuzzy rule is constant. All consequent membership functions are represented by singleton spikes. The Sugeno-style rule evaluation is shown in Figure 16.

*Example (cont.):*

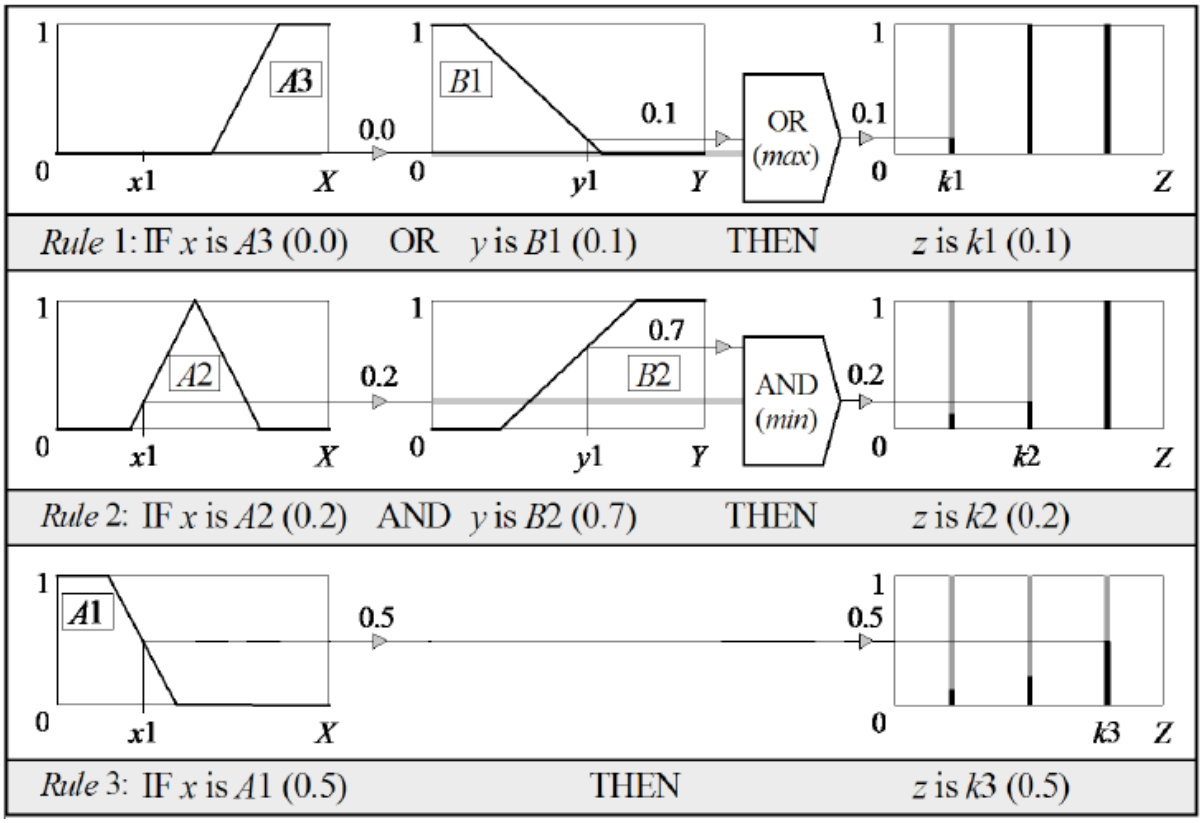



**Figure 16** Sugeno-style rule evaluation (adapted from http://www.4c.ucc.ie).

The Sugeno-style aggregation is shown in Figure 17.

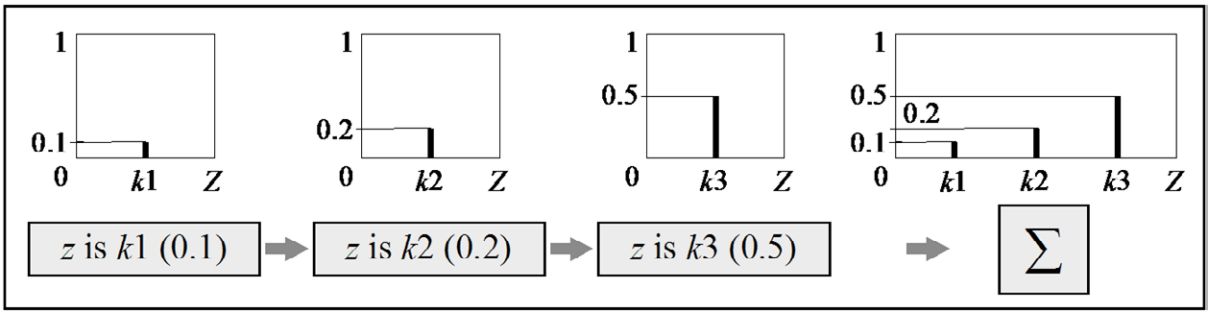



**Figure 17:** Sugeno-style aggregation (adapted from http://www.4c.ucc.ie).

We can find a weighted average (WA):

$$WA = \frac{(0.1\times 20)+(0.2\times 50)+(0.5\times 80)}{0.1+0.2+0.5} = 65$$

The Sugeno-style defuzzification is shown in Figure 18.

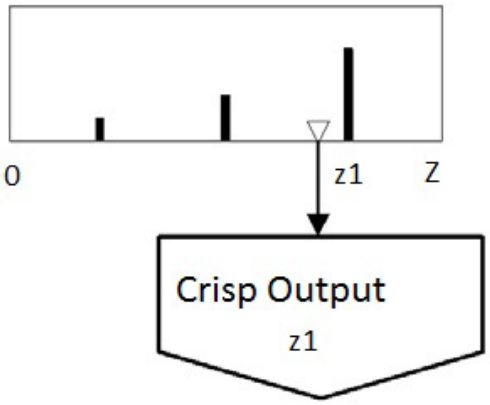



**Figure 18:** Sugeno-style defuzzification (adapted from http://www.4c.ucc.ie).

Mamdani or Sugeno? How to decide?

- The Mamdani method is widely accepted for capturing expert knowledge. It allows us to describe the expertise in more intuitive, more human-like manner. However, the Mamdani-type fuzzy inference entails a substantial computational burden.
- The Sugeno method is computationally efficient and works well with optimisation and adaptive techniques, which makes it very attractive in control problems, particularly for dynamic nonlinear systems.

## 2.2 Fuzzy control

Fuzzy control is considered to be the most successful area of application of the fuzzy set theory and fuzzy logic. Fuzzy controllers revolutionized the field of control engineering by their ability to perform process control by the utilization of human knowledge, thus enabling solutions to control problems for which mathematical models may not exist, or may be too difficult or computationally too expensive to construct.

### 2.2.1 Components of a fuzzy system

Figure 19 shows components of a fuzzy system. The input signals combined to the vector $x = (x_1, x_2, \ldots x_q)$ are crisp values, which are transformed into fuzzy sets in the fuzzification block. The output $u$ comes out directly from the defuzzification block, which transforms an output fuzzy set back to a crisp value. The set of membership functions responsible for the transforming part and the rule base as the relational part contain as whole modelling information about the system, which is processed by the inference machine. This rule-based fuzzy system is the basis of a fuzzy controller.

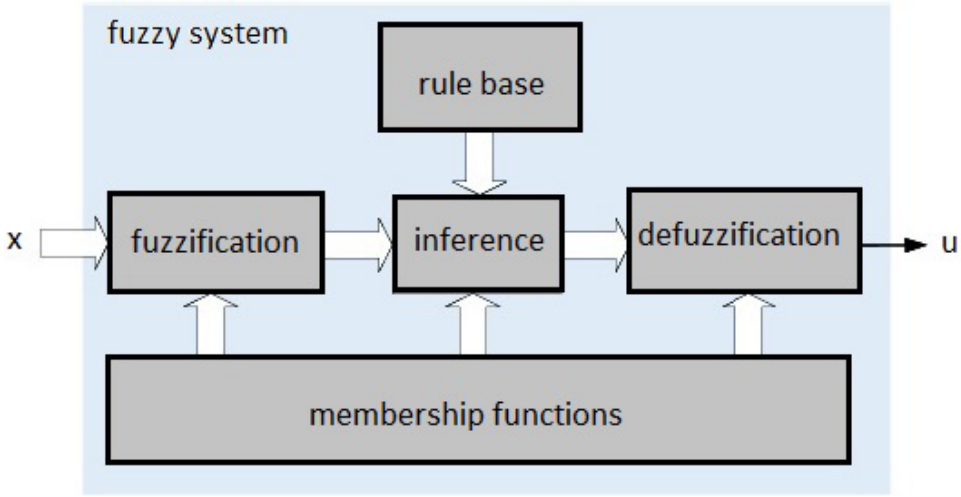



**Figure 19:** Components of a fuzzy system (adapted from http://www.atp.ruhr-uni-bochum.de).

### 2.2.2 Basic structure of a fuzzy controller

A fuzzy controller can be handled as a system that transmits information like a conventional controller with inputs containing information about the plant to be controlled and an output that is the manipulated variable. From outside, there is no vague information visible, both the input and output values are crisp values. The input values of a fuzzy controller consist of measured values from the plant that are either plant output values or plant states, or control errors derived from set-point values and controlled variables.

A control law represented in the form of a fuzzy system is a static control law. This means that the fuzzy rule-based representation of a fuzzy controller does not include any dynamics, which makes a fuzzy controller a static transfer element, like the standard state-feedback controller. In addition to this, a fuzzy controller is in general a fixed nonlinear static transfer element, which is due to those computational steps of its computational structure that have nonlinear properties. The computational structure of a fuzzy controller is described by presenting the computational steps involved. The computational structure of a fuzzy controller consists of three main steps as illustrated by the three blocks in Figure 20:

- signal conditioning and the input filter,
- the fuzzy system,
- signal conditioning and the output filter.

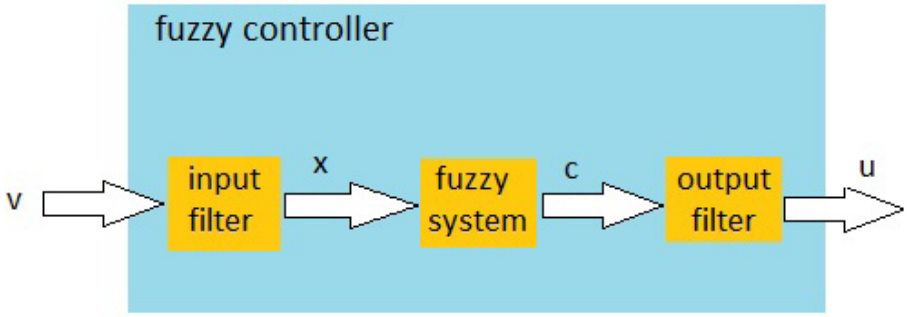



**Figure 20:** Basic structure of a fuzzy controller (adapted from http://www.atp.ruhr-uni-bochum.de).

Input and output filters implement signal conditioning. External input signals *v* must be scaled such that they can be fed as signals *x* into the fuzzification part of the fuzzy system. In many cases, the signals *v* are the control error *e* and its derivative is *e'*. In this case the input filter contains a differentiating element. Also other dynamical elements can be included in the input filter, e.g. integrators for the control error. Additionally, auxiliary signals from plant measurements may be used to represent plant states or disturbances acting on the plant. The design of this input filter depends on the application.

The fuzzy system contains a control strategy and consists of several components, see Figure 19. For example, a linguistic formulation of a proportional control strategy would be expressed by the following rules of the fuzzy system:

1. IF (control error positive) THEN (manipulated variable positive),
2. IF (control error zero) THEN (manipulated variable zero),
3. IF (control error negative) THEN (manipulated variable negative).

A proper rule base can be found either by asking experts or by evaluation of measurement data using data mining methods.

The output filter is used for adaptation of the crisp output from the fuzzy system concerning variable control. In principle, there are many possible dynamical and static operations. Often, the output of the fuzzy system describes an increment of the manipulated variable, and thus an integration of this increment must occur.

**A typical Fuzzy controller consists of four modules:**

The rule base, the inference engine, the fuzzification, and the defuzzification.

A typical Fuzzy Control algorithm would proceed as follows:

1. Obtaining information: To collect measurements of all relevant variables.
2. Fuzzification: To convert obtained measurements into appropriate fuzzy sets to capture uncertainties in the measurements.
3. Running the Inference Engine: To use fuzzified measurements to evaluate control rules in the rule base and select the set of possible actions.
4. Defuzzification: To convert the set of possible actions into a single numerical value.
5. The Loop: Go to step one.

Several defuzzification techniques have been proposed. The most common defuzzification methods are: the centre of gravity, the centre of maxima, the mean of maxima, and the root sum square.

*Example:*

The Figures below illustrate the notion of a simple fuzzy rule with one input and one output applied to the problem of an air motor speed controller for air conditioning. The Rules are given. Let us say the temperature is 22 degrees. This temperature is "right" to a degree of 0.6 and "warm" to a degree of 0.2 and it belongs to all others to a degree of zero. These rules are shown in Figure 21. The rule responses are shown in Figure 22 (thick lines).

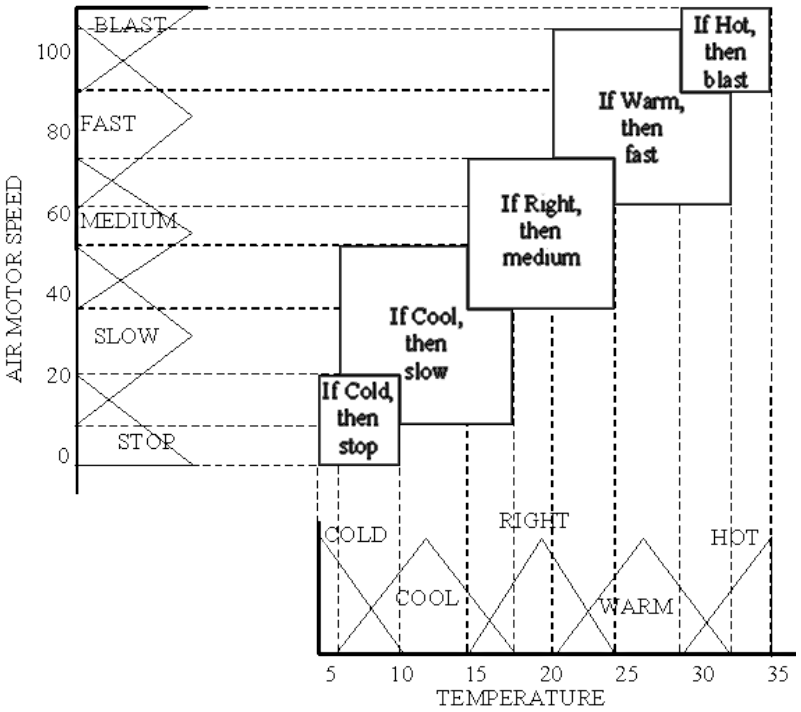



**Figure 21:** Air motor speed controller. Temperature (input) and speed (output) are fuzzy variables used in the set of rules (adapted from http://www.data-machine.nl/fuzzy1.htm).

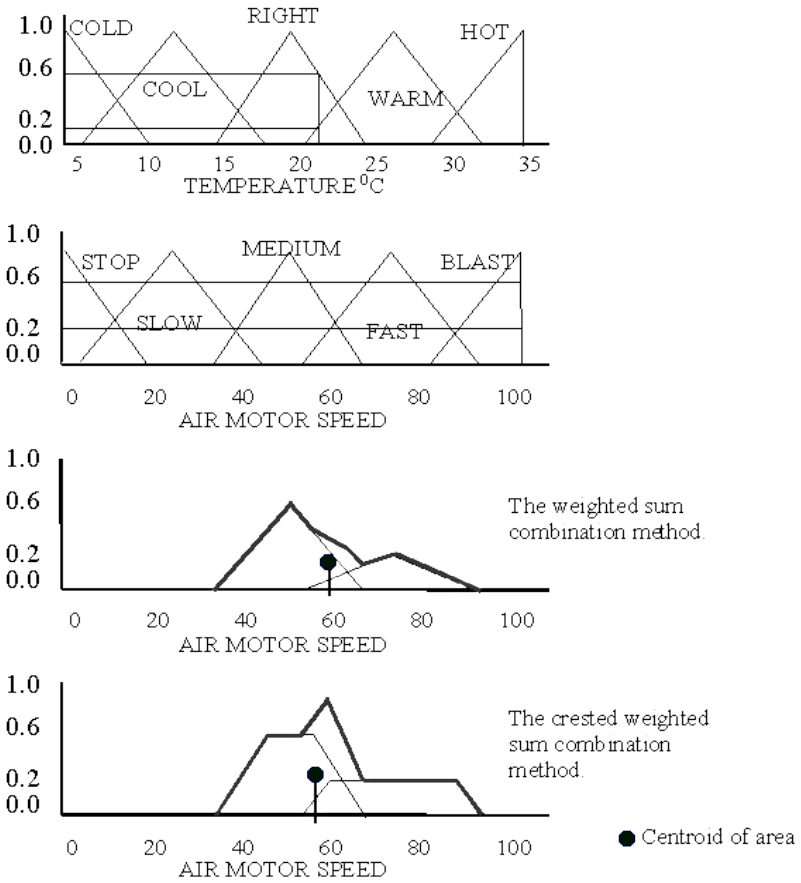



**Figure 22:** The temperature of 22 deg. "fires" two fuzzy rules. The resulting fuzzy value for air motor speed is defuzzified (adapted from http://www.data-machine.nl/fuzzy1.htm).

### 2.2.3 Representation using 2D characteristics

To provide the first approach for the design of membership functions for a fuzzy controller component, some prototype membership functions are introduced. For example, the following seven terms can be used to characterise the triangular shaped fuzzy sets according to Figure 23:

- NL negative large,
- NM negative medium,
- NS negative small
- AZ approximately zero
- PL positive large,
- PM positive medium,
- PS positive small.

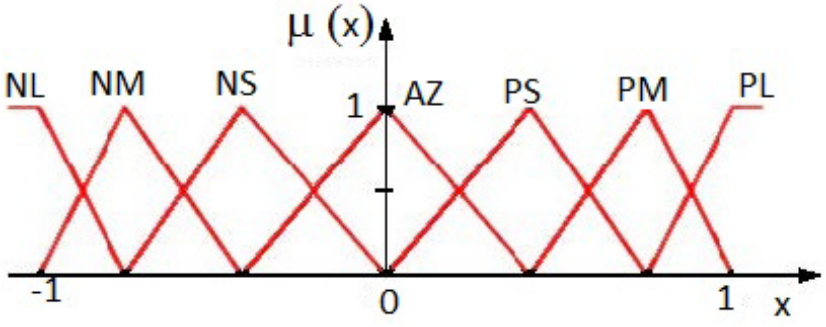



**Figure 23:** Prototype membership functions for a fuzzy set with seven linguistic terms (adapted from http://www.atp.ruhr-uni-bochum.de)

It is important to recognise that the fuzzy sets defined in Figure 23 and the seven linguistic terms are only a reasonable example. For various reasons, emerging from specific applications, other shapes of membership functions might be used over the given ranges. Moreover, different fuzzy sets may be defined for different variables. The prototype membership functions are usually chosen only as a preliminary candidate. They may later be modified by the designer.

For a fuzzy system using this kind of prototype with membership functions which overlaps, both for the premise and the conclusion the transfer characteristic is nonlinear, as shown in the following example.

Example:

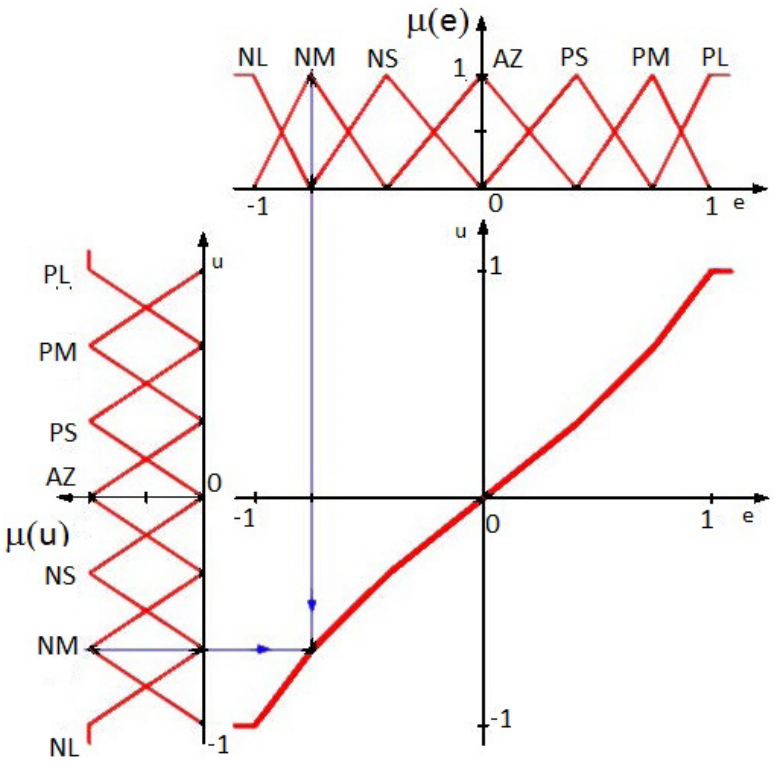



**Figure 24:** Nonlinear characteristic of a fuzzy controller (adapted from http://www.atp.ruhr-uni-bochum.de)

For a proportional fuzzy controller with the control error $e = w - y$ as input, with the manipulated variable $u$ as output, with the rule base and with both membership functions of the form shown in Figure 23, the static nonlinear characteristic $u = u(e)$ is as shown in Figure 24.

1) IF e = NL THEN u = NL
2) IF e = NM THEN u = NM
3) IF e = NS THEN u = NS
4) IF e = AZ THEN u = AZ
5) IF e = PS THEN u = PS
6) IF e = PM THEN u = PM
7) IF e = PL THEN u = PL

### 2.2.4 Influence of the membership functions and rule base on the characteristic

The above example shows that a fuzzy controller is a nonlinear controller. The form of the characteristic depends only on the rule base and the membership functions of *e* and *u*. In the following discussions about the influence of membership functions the following assumptions for the fuzzy controller with the input signal *e* and the output signal *u* will be used:

- For AND connectives the *min* and for OR connectives the *max* operator will be used.
- The max/min inference will be used.
- The defuzzification will be performed by COG method with symmetrical membership functions at the margins.

Input and output values are normalised to the interval [-1, 1], and at first, only the following three linguistic terms NS (negative small), AZ (approximate zero) and PS (positive small) are considered. The rule base of a proportional fuzzy controller is the following:

1) IF e = NS THEN u = NS
2) IF e = AZ THEN u = AZ
3) IF e = PS THEN u = PS

The membership functions are shown in Figure 25a and 25b. The static characteristic (Figure 25c) is odd symmetrical about the origin due to the symmetry of the membership functions. Because of the different supports of the membership function for the fuzzy sets AZ of both functions, the characteristic is approximately piecewise linear and has three distinct levels. The membership functions of the input *e* have two overlaps in the intervals [-0.6, -0.4] and [0.4, 0.6] that correspond precisely with the ranges with the positive slope of the curve. The reason for this is just that two rules in these ranges are simultaneously active. On the other hand, in the non-overlapping ranges only one rule is active. The membership function of the output depends in this case only on the degree of relevance and thus the centre of gravity of the membership function remains constant.

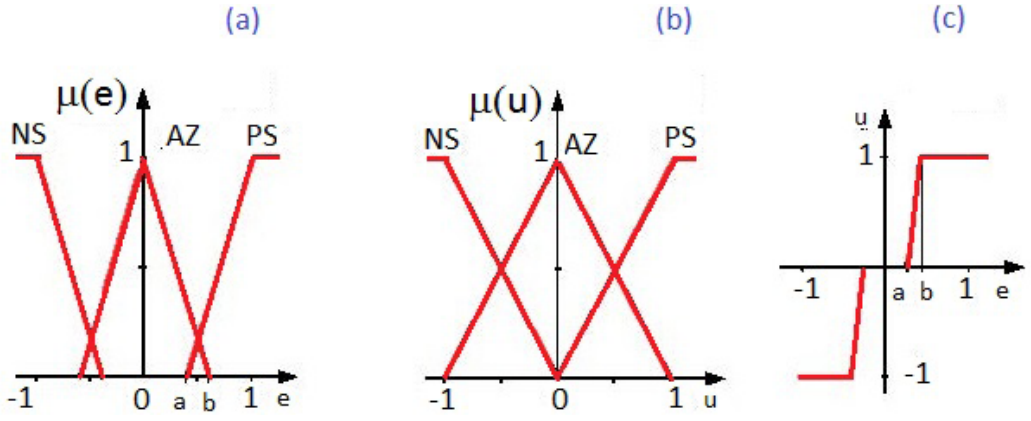



**Figure 25:** Membership functions and static characteristic of the fuzzy controller (adapted from http://www.atp.ruhr-uni-bochum.de)

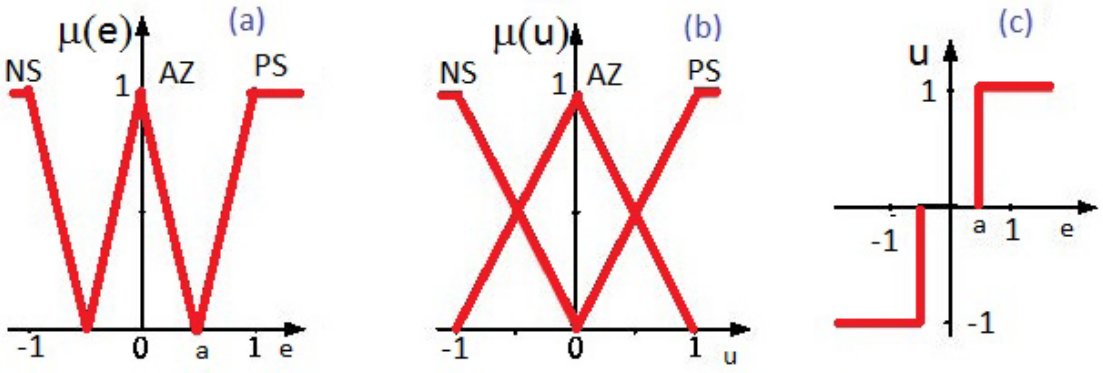



**Figure 26:** Influence of the (c) characteristic of a proportional fuzzy controller (a) without overlapping in the input membership functions and (b) with full overlapping in the output membership functions (adapted from http://www.atp.ruhr-uni-bochum.de)

If the number of linguistic terms for the input and output is increased, the characteristic is similar, but with more sections. The number of sections depends only on the number of linguistic terms and the width of the sections depends on the degree of overlapping. In the special case without overlapping in the input one obtains the characteristic of a three-level controller, as shown in Figure 26. In this case only one rule is active such that only the three crisp values -1, 0 and 1 are generated. Next, we consider varying the degree of overlap of the output membership functions. Figure 27 shows the case with full overlap on input and output, where the result is approximately a linear behaviour.

A modification of the output membership functions so that they do not overlap will cause the characteristic to become close to that of Figure 27, compare with Figure 28. Therefore one can establish the fact that the degree of overlap in the input membership functions has a strong influence on the static characteristic of a fuzzy controller. While small overlaps in the input membership functions generate step characteristics, with a higher degree of overlap the curves become smoother. The influence of overlap in the output membership functions has less effect on the characteristic. For a reduction of the support of the output membership functions the characteristic of Figure 29 is obtained, which does not differ significantly from that of Figure 27.

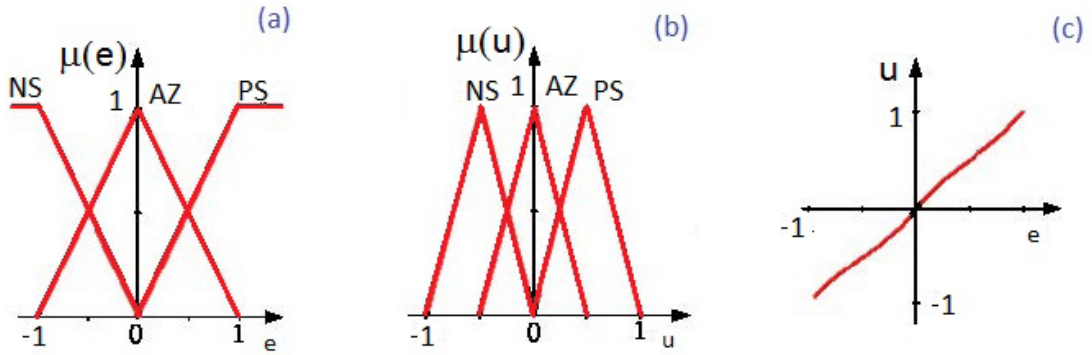



**Figure 27:** Influence on the (c) characteristic of a proportional fuzzy controller with (a) full overlap in the input membership functions and (b) full overlap in the output membership functions (adapted from http://www.atp.ruhr-uni-bochum.de)

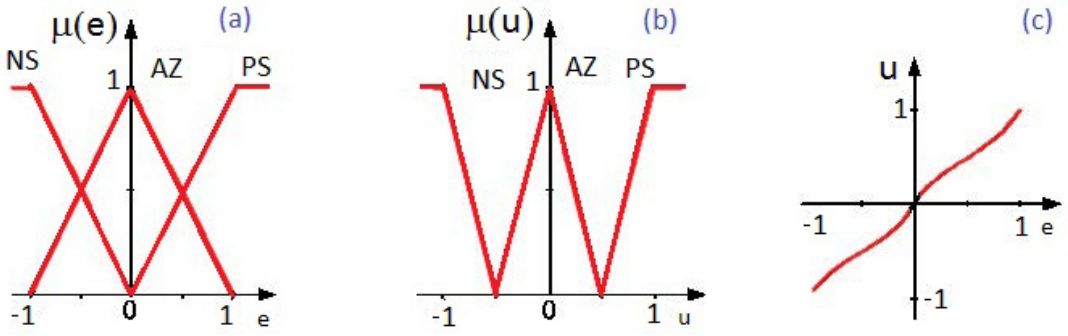



**Figure 28:** Influence on the (c) characteristic of a proportional fuzzy controller with (a) full overlap in the input membership functions and with (b) reduced support in the output membership functions (adapted from http://www.atp.ruhr-uni-bochum.de)

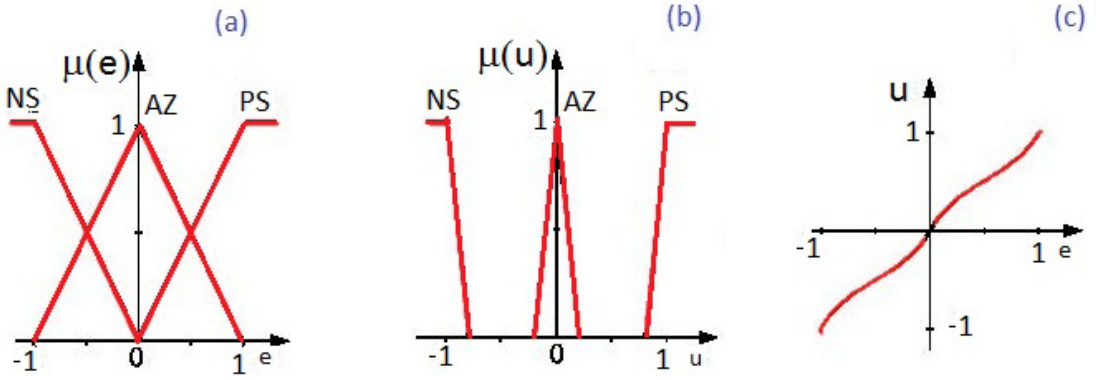



**Figure 29:** Influence on the (c) characteristic of a proportional fuzzy controller with (a) full overlap in the input membership functions and with (b) reduced support in the output membership functions (adapted from http://www.atp.ruhr-uni-bochum.de)

The size of the individual output membership function has a strong influence on the characteristic. Figure 30 shows the case for a very small support of the output membership function AZ, which generates a S-type characteristic with a high gain at the origin. Widening the support of the membership function AZ inverts the S-curve with a small gain at the origin, as shown in Figure 31. Thus, the form of the characteristic depends strongly on the support of the individual output membership function.

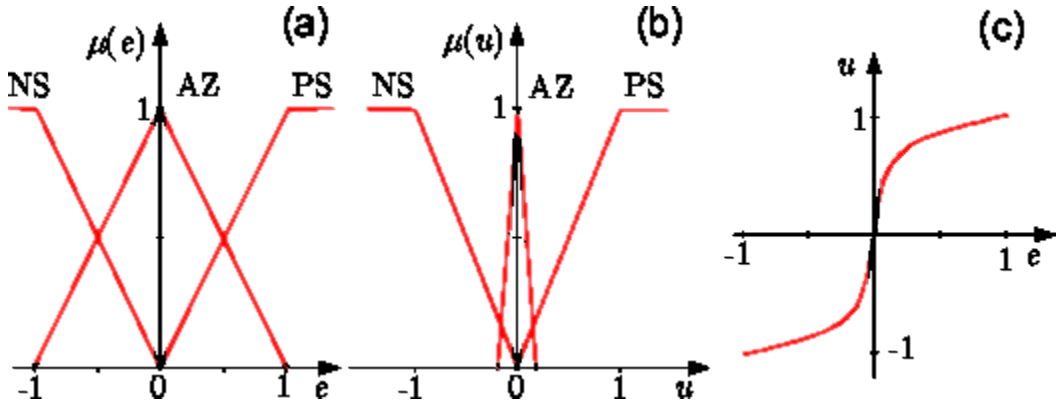



**Figure 30:** Influence on the (c) characteristic of a proportional fuzzy controller with (a) full overlap in the input membership functions and with (b) a small support in the output membership function AZ (adapted from http://www.atp.ruhr-uni-bochum.de)

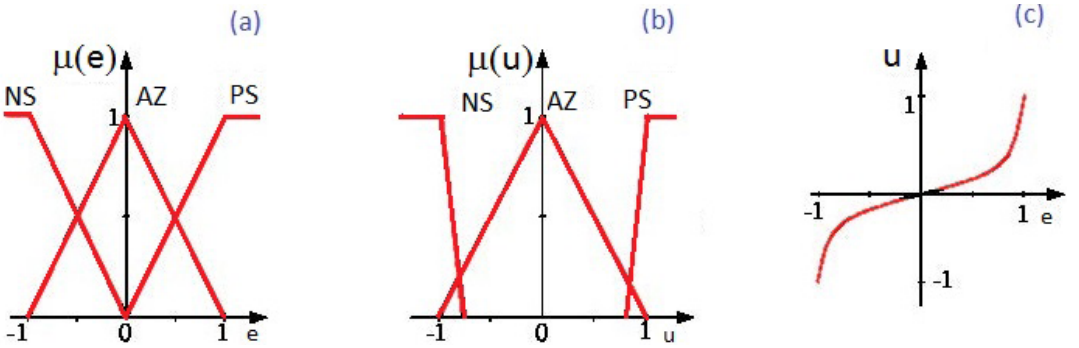



**Figure 31:** Influence on the (c) characteristic of a proportional fuzzy controller with (a) full overlap in the input membership functions and with (b) a large support in the output membership function AZ (adapted from http://www.atp.ruhr-uni-bochum.de)

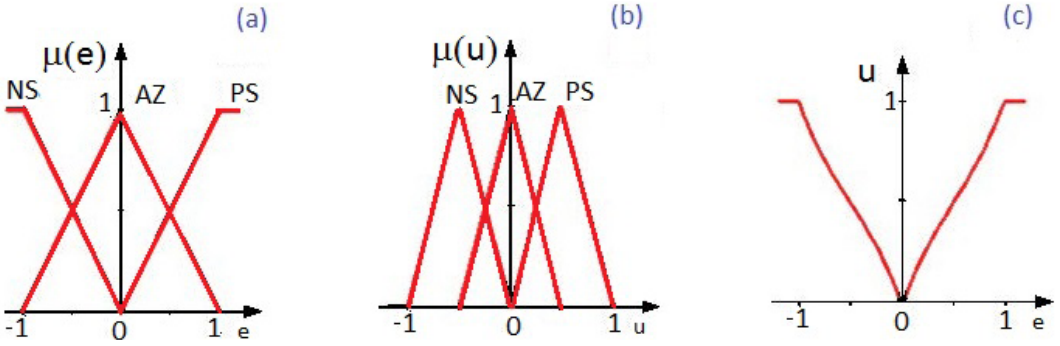



**Figure 32:** Influence on the rule base on the (c) characteristic of a proportional fuzzy controller with (a) full overlap in the input membership functions and (b) full overlap in the output membership functions (adapted from http://www.atp.ruhr-uni-bochum.de)

The effects of a modified rule base will be demonstrated by an example. The same full overlapping membership functions are used as in Figure 27a and 27 b. A modified rule base of the form

(1) IF e = NS THEN u = PS
(2) IF e = AZ THEN u = AZ
(3) IF e = PS THEN u = NS

will give a modulus-type of characteristic, as shown in Figure 32.

### 2.2.5 Representation using 3D characteristics

Up to now, fuzzy controllers with only one input and one output of the fuzzy system have been considered. The same arguments with respect to the degrees of freedom of a fuzzy system are also valid in the case of multiple inputs. A graphical representation of the characteristic is not as easy as in the 2D cases. Moreover, for the case of two inputs, a 3D representation is possible. For a fuzzy controller with a fuzzy system having two inputs $e_1$ and $e_2$ and one output $u$ one gets a band of characteristics in a 2D discrete representation or a 3D representation with the output over the two inputs, as shown in Figure 33 for a PD-type of fuzzy controller.

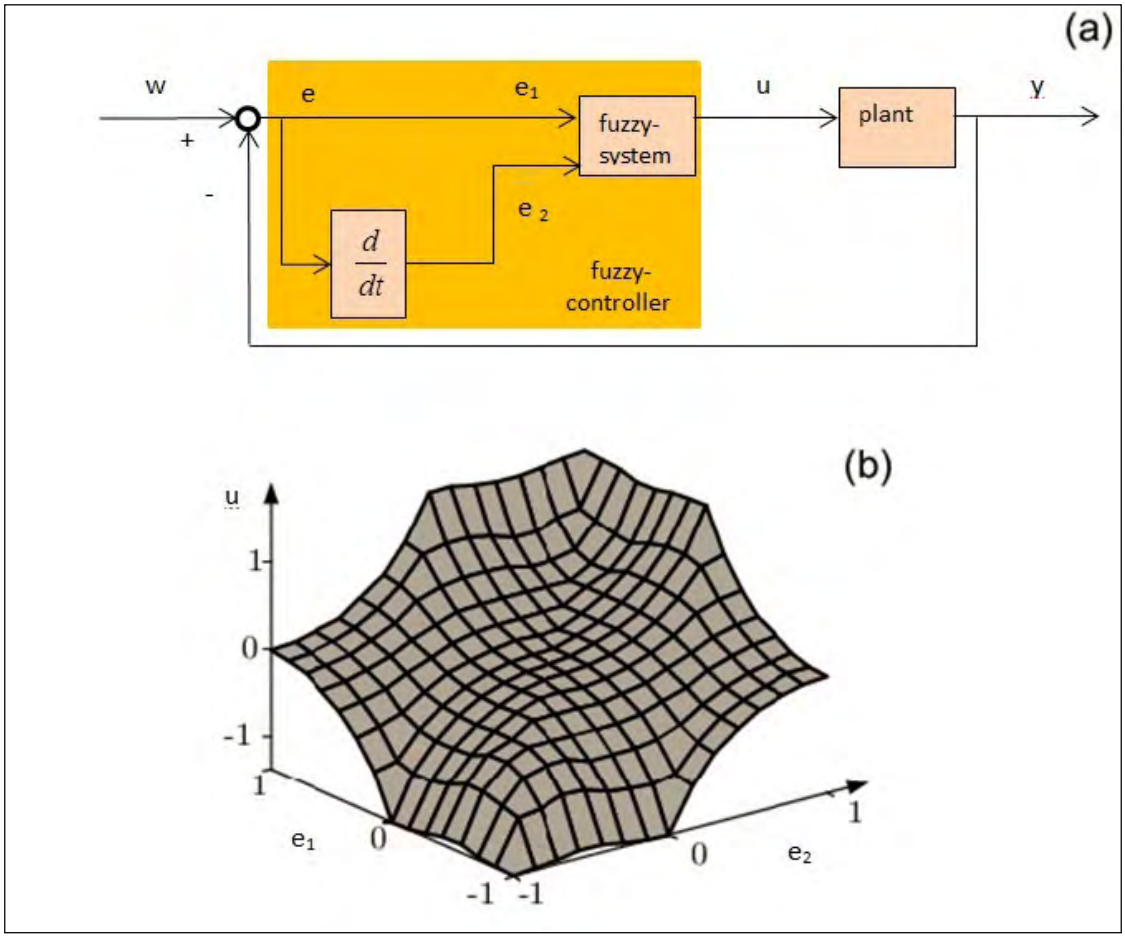



**Figure 33:** Control system with PD-type fuzzy controller: (a) block diagram and (b) 3D representation of the characteristics of the fuzzy system with $e_1$ = e and $e_2 = e'$ (adapted from http://www.atp.ruhr-uni-bochum.de)

For the case with more than two inputs, projections on the 3D space can be used to generate multiple 3D diagrams, but in general these representations have only a limited usefulness. A fuzzy controller is typically a classical controller using nonlinear characteristics. But the design and parameterisation is entirely different.

### Example of a fuzzy control system

In the following the design and functioning of a fuzzy control system will be presented using the example of the portal-type loading crane shown in Figure 34.

**Figure 34**: View of a portal-type loading crane (adapted from http://www.atp.ruhr-uni-bochum.de)

# 3 Evolutionary Computing

Evolutionary Computing is the collective name for a range of problem-solving techniques based on principles of biological evolution, such as natural selection and genetic inheritance. These techniques are being increasingly widely applied to a variety of problems, ranging from practical applications in industry and commerce to leading-edge scientific research.

In computer science, evolutionary computation is a subfield of artificial intelligence (more particularly computational intelligence) that involves combinatorial optimization problems. Evolutionary computation uses iterative progress, such as growth or development in a population. This population is then selected in a guided random search using parallel processing to achieve the desired end. Such processes are often inspired by biological mechanisms of evolution. As evolution can produce highly optimised processes and networks, it has many applications in computer science.

Problem solution using evolutionary algorithms is shown in Figure 35 (Pohlheim 2006).

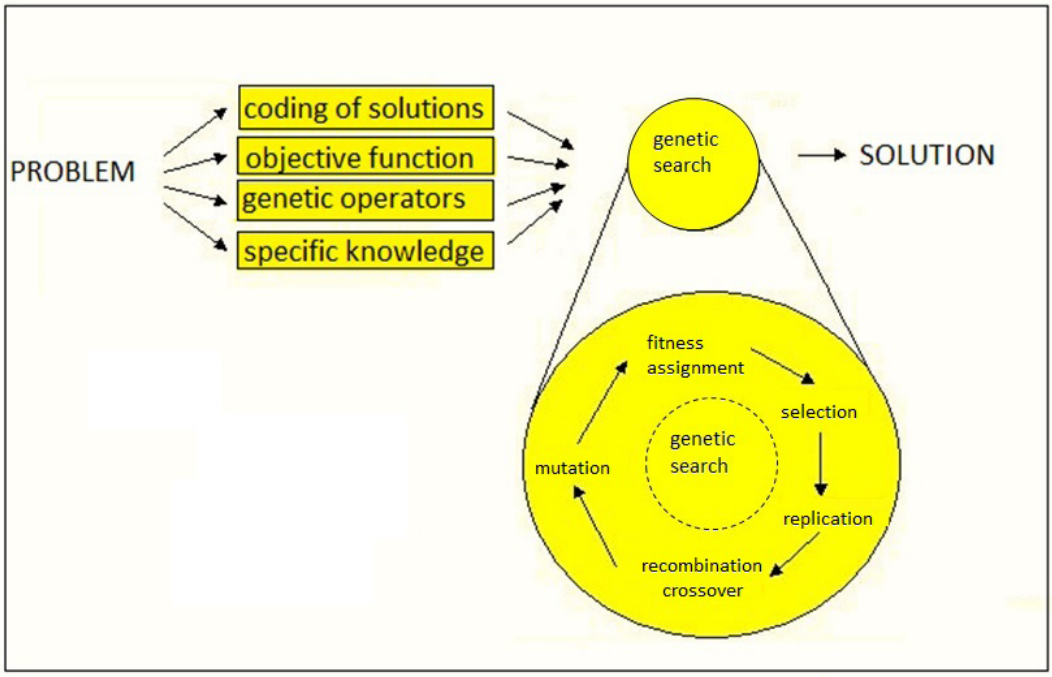



**Figure 35:** Problem solution using evolutionary algorithms (adapted from http://jpmc.sourceforge.net)

## 3.1 Evolutionary algorithms

Different main schools of evolutionary algorithms have been evolved during the last 50 years: genetic algorithms, mainly developed in the USA by J.H. Holland (Holland 1975), evolutionary strategies, developed in Germany by I. Rechenberg (Rechenberg 1973) and H.-P. Schwefel (Schwefel 1981), and evolutionary programming (Fogel, Owens, and Walsh 1966). Each of these constitutes represents a different approach, however, they are inspired by the same principles of natural evolution. A good introductory survey can be found in (Fogel 1994).

Evolutionary algorithms are stochastic search methods that mimic the metaphor of natural biological evolution. Evolutionary algorithms operate on a population of potential solutions applying the principle of survival of the fittest to produce better and better approximations to a solution. At each generation, a new set of approximations is created by the process of selecting individuals according to their level of fitness in the problem domain and breeding them together using operators borrowed from natural genetics. This process leads to the evolution of populations of individuals that are better suited to their environment than the individuals that they were created from, just as in natural adaptation. Evolutionary algorithms model natural processes, such as selection, recombination, mutation, migration, locality and neighbourhood. Figure 36 shows the structure of a simple evolutionary algorithm. Evolutionary algorithms work on populations of individuals instead of single solutions. In this way the search is performed in a parallel manner.

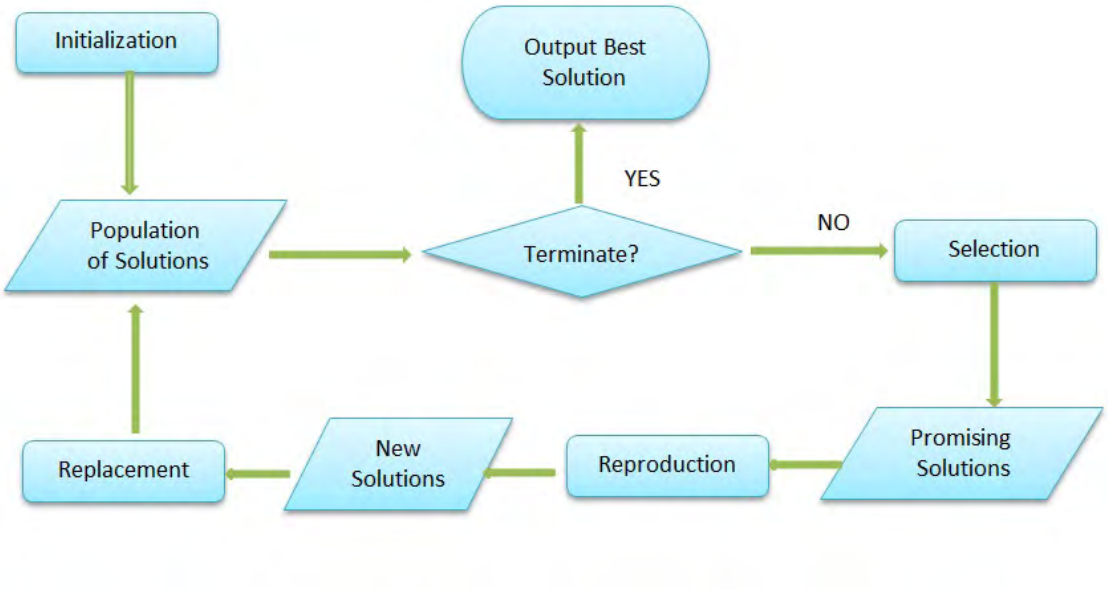



**Figure 36:** Structure of a single population evolutionary algorithm (adapted from www.sciencedirect.com)

At the beginning of the computation a number of individuals (the population) are randomly initialized. The objective function is then evaluated for these individuals. The first/initial generation is produced. If the optimization criteria are not met, the creation of a new generation starts. Individuals are selected according to their fitness for the production of offspring. Parents are recombined to produce offspring. All offspring will be mutated with a certain probability. The fitness of the offspring is then computed. The offspring are inserted into the population replacing the parents, producing a new generation. This cycle is performed until the optimization criteria are reached.

From the above discussion, it can be seen that evolutionary algorithms differ substantially from more traditional search and optimization methods. The most significant differences are (Pohlheim 2006):

- Evolutionary algorithms search a population of points in parallel, not just a single point.
- Evolutionary algorithms do not require derivative information or other auxiliary knowledge; only the objective function and corresponding fitness levels influence the directions of search.
- Evolutionary algorithms use probabilistic transition rules, not deterministic ones.
- Evolutionary algorithms are generally more straightforward to apply, because no restrictions for the definition of the objective function exist.
- Evolutionary algorithms can provide a number of potential solutions to a given problem. The final choice is left to the user. (Thus, in cases where the particular problem does not have one individual solution, for example a family of pareto-optimal solutions, as in the case of multi-objective optimization and scheduling problems, then the evolutionary algorithm is potentially useful for identifying these alternative solutions simultaneously.)

### 3.1.1 Selection

In selection the offspring producing individuals are chosen. The first step is fitness assignment. Each individual in the selection pool receives a reproduction probability depending on the own objective value and the objective value of all other individuals in the selection pool. This fitness is used for the actual selection step afterwards. Throughout the chapter some terms are used for comparing the different selection schemes. The definitions of these terms follow (Baker 1987).

*selective pressure*: probability of the best individual being selected compared to the average probability of selection of all individuals

*bias*: absolute difference between an individual's normalized fitness and its expected probability of reproduction

*spread:* range of possible values for the number of offspring of an individual

*loss of diversity:* proportion of individuals of a population that is not selected during the selection phase

*selection intensity:* expected average fitness value of the population after applying a selection method to the normalized Gaussian distribution

*selection variance:* expected variance of the fitness distribution of the population after applying a selection method to the normalized Gaussian distribution

**Roulette wheel selection**

The simplest selection scheme is roulette-wheel selection, also called stochastic sampling with replacement (Baker 1987). This is a stochastic algorithm and involves the following technique: The individuals are mapped to contiguous segments of a line, such that each individual's segment is equal in size to its fitness. A random number is generated and the individual whose segment spans the random number is selected. The process is repeated until the desired number of individuals is obtained (called mating population). This technique is analogous to a roulette wheel with each slice proportional in size to the fitness.

*Example*

Table 2 shows the selection probability for 11 individuals. Individual 1 is the most fit individual and occupies the largest interval, whereas individual 10 as the second least fit individual has the smallest interval on the line (see Figure 37). Individual 11, the least fit interval, has a fitness value of 0 and get no chance for reproduction.

| Number of individual | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| fitness value | 2.0 | 1.8 | 1.6 | 1.4 | 1.2 | 1.0 | 0.8 | 0.6 | 0.4 | 0.2 | 0.0 |
| selection probability | 0.18 | 0.16 | 0.15 | 0.13 | 0.11 | 0.09 | 0.07 | 0.06 | 0.03 | 0.02 | 0.0 |

**Table 2:** Selection probability and fitness value

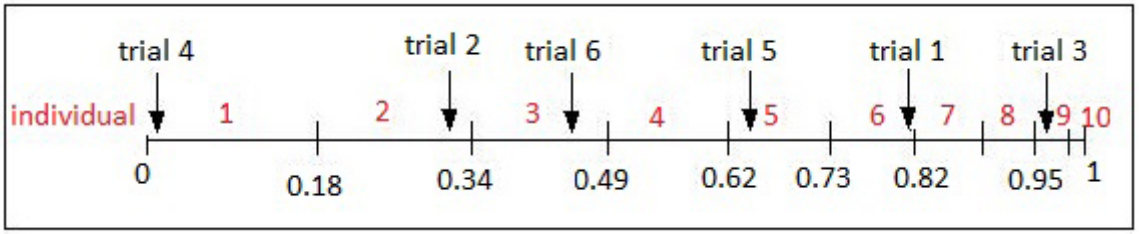



**Figure 37:** Roulette-wheel selection (adapted from http://www.geatbx.com/)

For selecting, the mating population the appropriate number of uniformly distributed random numbers (uniform distributed between 0.0 and 1.0) is independently generated.

*Sample of 6 random numbers:* 0.81, 0.32, 0.96, 0.01, 0.65, 0.42.

Figure 37 shows the selection process of the individuals for the example in Table 2 together with the above sample trials.

*After selection the mating population consists of the individuals*: 1, 2, 3, 5, 6, 9.

The roulette-wheel selection algorithm provides a zero bias but does not guarantee minimum spread.

**Stochastic universal sampling**

Stochastic universal sampling (Baker 1987) provides zero bias and minimum spread. The individuals are mapped to contiguous segments of a line, such that each individual's segment is equal in size to its fitness exactly as in roulette-wheel selection. Here equally spaced pointers are placed over the line as many as there are individuals to be selected. Consider *NPointer* the number of individuals to be selected, then the distance between the pointers are 1/*NPointer* and the position of the first pointer is given by a randomly generated number in the range [0, 1/*NPointer*].

*Example (cont.)*

For 6 individuals to be selected, the distance between pointers is 1/6 = 0.167. Figure 38 shows the selection for the above example.

*Sample of 1 random number in the range* [0, 0.167]: 0.1.

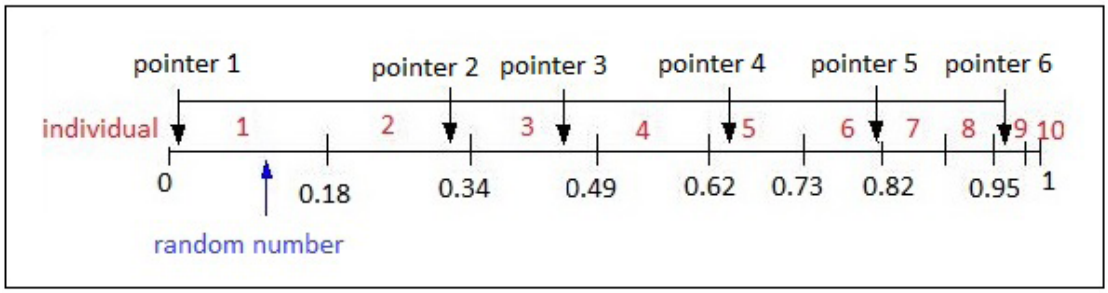



**Figure 38:** Stochastic universal sampling (adapted from http://www.geatbx.com/)

*After selection the mating population consists of the individuals*: 1, 2, 3, 4, 6, 8.

Stochastic universal sampling ensures a selection of offspring which is closer to what is deserved then roulette wheel selection.

**Local selection**

In local selection every individual resides inside a constrained environment called the local neighbourhood. (In the other selection methods the whole population or subpopulation is the selection pool or neighbourhood.) Individuals interact only with individuals inside this region. The neighbourhood is defined by the structure in which the population is distributed. The neighbourhood can be seen as the group of potential mating partners.

The first step is the selection of the first half of the mating population uniform at random (or using one of the other mentioned selection algorithms, for example, stochastic universal sampling or tournament selection). Now a local neighbourhood is defined for every selected individual. Inside this neighbourhood the mating partner is selected (best, fitness proportional, or uniform at random).

The structure of the neighbourhood can be:

- *linear*
  full ring, half ring (see Figure 39)

- *two-dimensional*
  full cross, half cross (see Figure 40, left)
  full star, half star (see Figure 40, right)

- *three-dimensional* and more complex with any combination of the above structures.

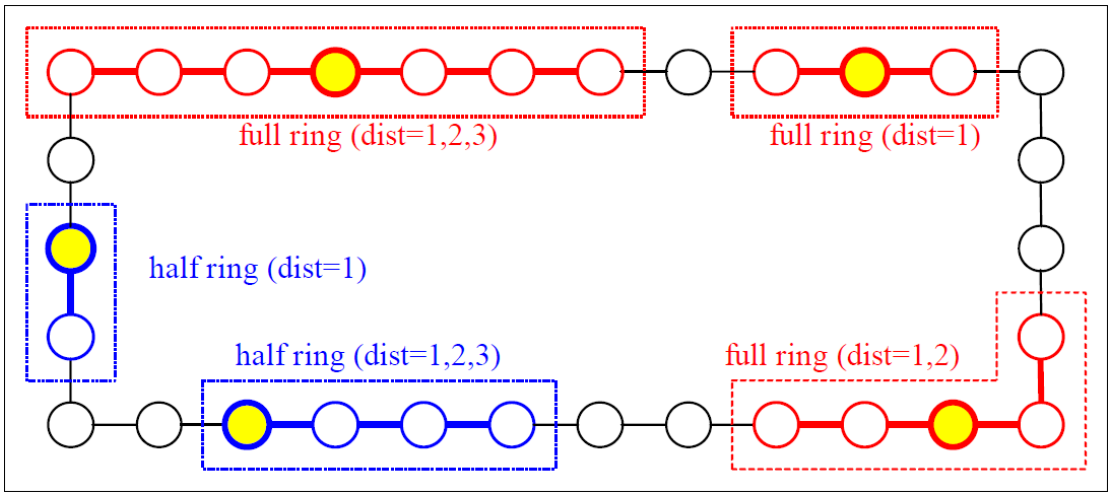



**Figure 39:** Linear neighbourhood: full and half ring (adapted from http://www.geatbx.com/)

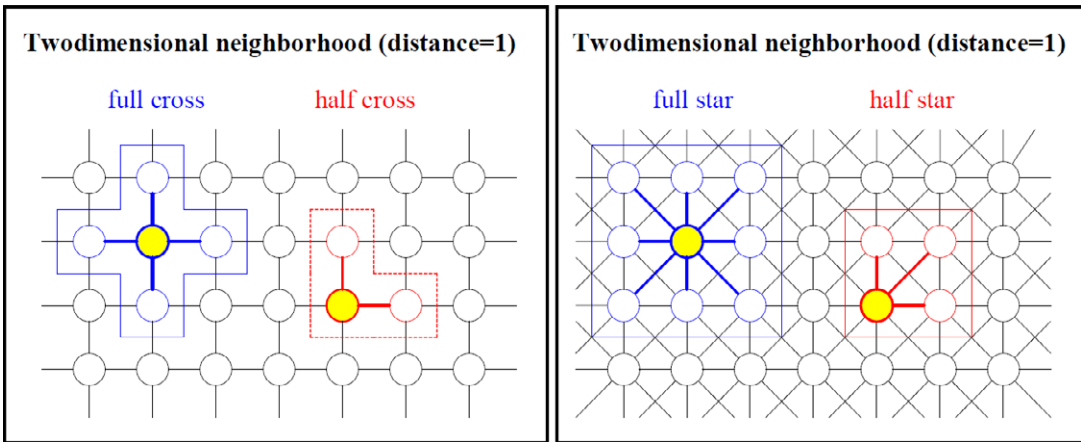



**Figure 40:** Two-dimensional neighbourhood; left: full and half cross, right: full and half star (adapted from http://www.geatbx.com/)

The distance between possible neighbours together with the structure determines the size of the neighbourhood. Between individuals of a population an "isolation by distance" exists. The smaller the neighbourhood, the bigger the isolation distances. However, because of overlapping neighbourhoods, propagation of new variants takes place. This assures the exchange of information between all individuals. The size of the neighbourhood determines the speed of propagation of information between the individuals of a population, thus deciding between rapid propagation and maintenance of a high diversity/variability in the population. A higher variability is often desired, thus preventing problems such as premature convergence to a local minimum. Local selection in a small neighbourhood performed better than local selection in a bigger neighbourhood. Nevertheless, the interconnection of the whole population must still be provided. Two-dimensional neighbourhood with structure half star using a distance of 1 is recommended for local selection. However, if the population is bigger (>100 individuals) a greater distance and/or another two-dimensional neighbourhood should be used (Pohlheim 2006).

### Tournament selection

In tournament selection (Goldberg and Deb 1991) a number *Tour* of individuals is chosen randomly from the population and the best individual from this group is selected as parent. This process is repeated as often as individuals must be chosen. These selected parents produce uniform at random offspring. The parameter for tournament selection is the tournament size *Tour*. *Tour* takes values ranging from 2 to *Nind* (number of individuals in population). Table 3 and Figure 41 show the relation between the tournament size and selection intensity (Blickle 1995).

| tournament size | 1 | 2 | 3 | 5 | 10 | 30 |
|---|---|---|---|---|---|---|
| selection intensity | 0 | 0.56 | 0.85 | 1.15 | 1.53 | 2.04 |

**Table 3:** Relation between tournament size and selection intensity

In (Blickle 1995) an analysis of tournament selection can be found.

Selection intensity

$$SelInt_{Turnier}(Tour) \approx \sqrt{2 \cdot \left(\ln(Tour) - \ln\left(\sqrt{4.14 \cdot \ln(Tour)}\right)\right)}$$

Loss of diversity

$$LossDiv_{Turnier}(Tour) = Tour^{\frac{-1}{Tour-1}} - Tour^{\frac{-Tour}{Tour-1}}$$

(About 50% of the population are lost at tournament size *Tour*=5).

Selection variance

$$SelVar_{Turnier}(Tour) \approx \frac{0.918}{\mathrm{h}\,(1.186 + 1.328 \cdot Tour)}$$

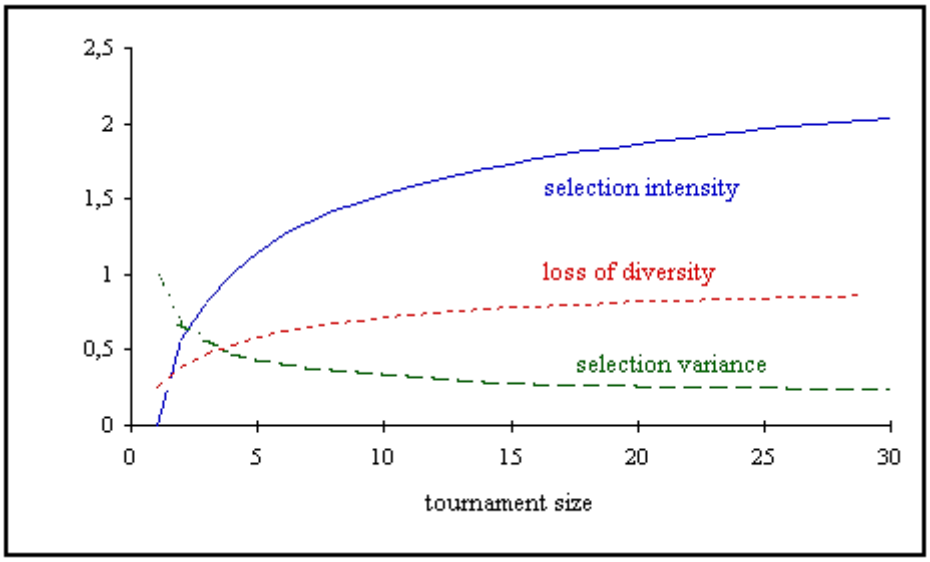



**Figure 41:** Properties of tournament selection (adapted from http://www.geatbx.com/)

### 3.1.2 Recombination

Recombination produces new individuals in combining the information contained in two or more parents (parents – mating population). This is done by combining the variable values of the parents. Depending on the representation of the variables different methods must be used.

The methods for binary valued variables constitute special cases of the discrete recombination. These methods can be applied to integer valued and real valued variables as well.

**Discrete recombination – All representations**

Discrete recombination (Mühlenbein and Schlierkamp-Voosen 1993) performs an exchange of variable values between individuals. For each position, the parent who contributes its variable to the offspring is chosen randomly with equal probability.

$$Var_i^0 = Var_i^{P_1} \cdot a_i + Var_i^{P_2} \cdot (1 - a_i) \quad i \in (1,2,\cdots, Nvar),$$
$$a_i \in \{0,1\} \quad \text{uniform at random, } a_i \quad \text{for each } i \text{ new defined}$$

Discrete recombination generates corners of the hypercube defined by the parents. Figure 42 shows the geometric effect of discrete recombination.

*Example*

Consider the following two individuals with 3 variables each (3 dimensions), which will also be used to illustrate the other types of recombination for real valued variables:

*individual 1* 12 25 5
*individual 2* 123 4 34

For each variable the parent who contributes its variable to the offspring is chosen randomly with equal probability:

*sample 1* 2 2 1
*sample 2* 1 2 1

After recombination the new individuals are created:

*offspring 1* 123 4 5
*offspring 2* 12 4 5

Discrete recombination can be used with any kind of variables (binary, integer, real or symbols).

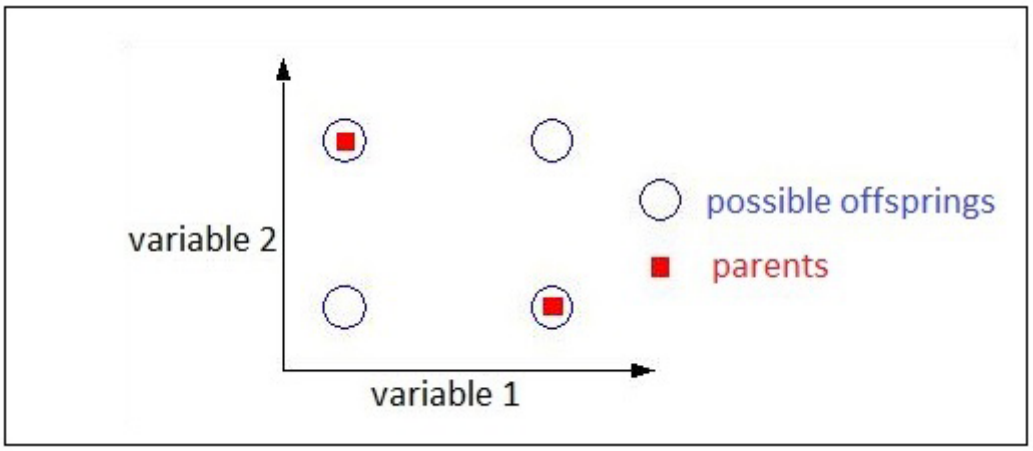



**Figure 42:** Possible positions of the offspring after discrete recombination (adapted from http://www.geatbx.com/)

**Intermediate recombination – Real valued recombination**

Intermediate recombination (Mühlenbein and Schlierkamp-Voosen 1993) is a method only applicable to real variables (and not binary variables). Here, the variable values of the offspring are chosen somewhere around and between the variable values of the parents.

Offspring are produced according to the rule:

$$Var_i^0 = Var_i^{P_1} \cdot a_i + Var_i^{P_2} \cdot (1 - a_i) \quad i \in (1,2,\cdots,Nvar),$$
$$a_i \in [-d, 1+d] \text{ uniform at random, } d = 0.25, \quad a_i \text{ for each } i \text{ new}$$

where $a$ is a scaling factor chosen uniformly at random over an interval $[-d, 1+d]$ for **each** variable anew.

The value of the parameter $d$ defines the size of the area for possible offspring. A value of $d = 0$ defines the area for offspring the same size as the area spanned by the parents. This method is called (standard) intermediate recombination. Because most variables of the offspring are not generated on the border of the possible area, the area for the variables shrinks over the generations. This shrinkage occurs just by using (standard) intermediate recombination. This effect can be prevented by using a larger value for $d$. A value of $d = 0.25$ ensures (statistically), that the variable area of the offspring is the same as the variable area spanned by the variables of the parents. See Figure 43 for a picture of the area of the variable range of the offspring defined by the variables of the parents.

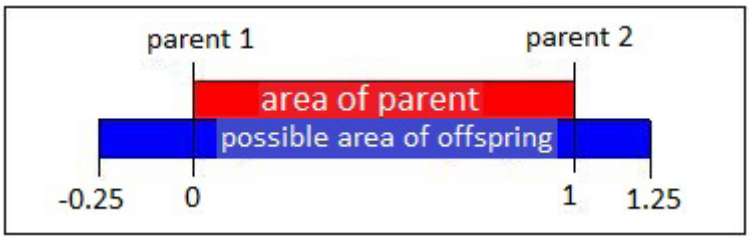



**Figure 43:** Area for variable value of offspring compared to parents in intermediate recombination (adapted from http://www.geatbx.com/)

*Example*

Consider the following two individuals with 3 variables each:

| | | | |
|---|---|---|---|
| *individual1* | 12 | 25 | 5 |
| *individual 2* | 123 | 4 | 34 |

The chosen $a$ for this example are:

| | | | |
|---|---|---|---|
| *sample 1* | 0.5 | 1.1 | -0.1 |
| *sample 2* | 0.1 | 0.8 | 0.5 |

The new individuals are calculated as:

| | | | |
|---|---|---|---|
| *offspring 1* | 67.5 | 1.9 | 2.1 |
| *offspring 2* | 23.1 | 8.2 | 19.5 |

Intermediate recombination is capable of producing any point within a hypercube slightly larger than that defined by the parents. Figure 44 shows the possible area of offspring after intermediate recombination.

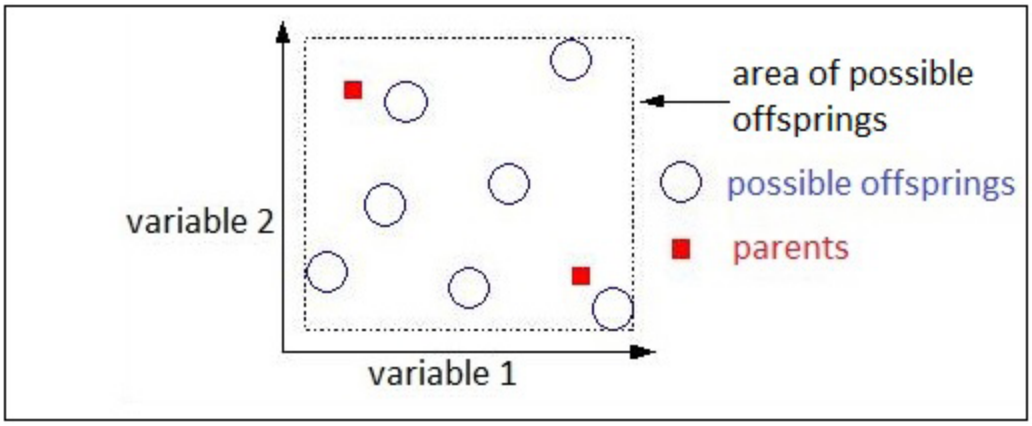



**Figure 44:** Possible area of the offspring after intermediate recombination (adapted from http://www.geatbx.com/)

**Line recombination – Real valued recombination**

Line recombination (Mühlenbein and Schlierkamp-Voosen 1993) is similar to intermediate recombination, except that only one value of $a$ for all variables is used. The same $a$ is used for **all** variables:

$$Var_i^0 = Var_i^{P_1} \cdot a_i + Var_i^{P_2} \cdot (1 - a_i) \quad i \in (1,2,\cdots,Nvar),$$
$$a_i \in [-d, 1+d] \text{ uniform at random, } d = 0.25, \quad a_i \text{ for all } i \text{ identical}$$

For the value of $d$ the statements given for intermediate recombination are applicable.

*Example*

Consider the following two individuals with 3 variables each:

| | | | |
|---|---|---|---|
| *individual 1* | 12 | 25 | 5 |
| *individual 2* | 123 | 4 | 34 |

The chosen *a* for this example are:

| | |
|---|---|
| *sample 1* | 0.5 |
| *sample 2* | 0.1 |

The new individuals are calculated as:

| | | | |
|---|---|---|---|
| *offspring 1* | 67.5 | 14.5 | 19.5 |
| *offspring 2* | 23.1 | 22.9 | 7.9 |

Line recombination can generate any point on the line defined by the parents. Figure 45 shows the possible positions of the offspring after line recombination.

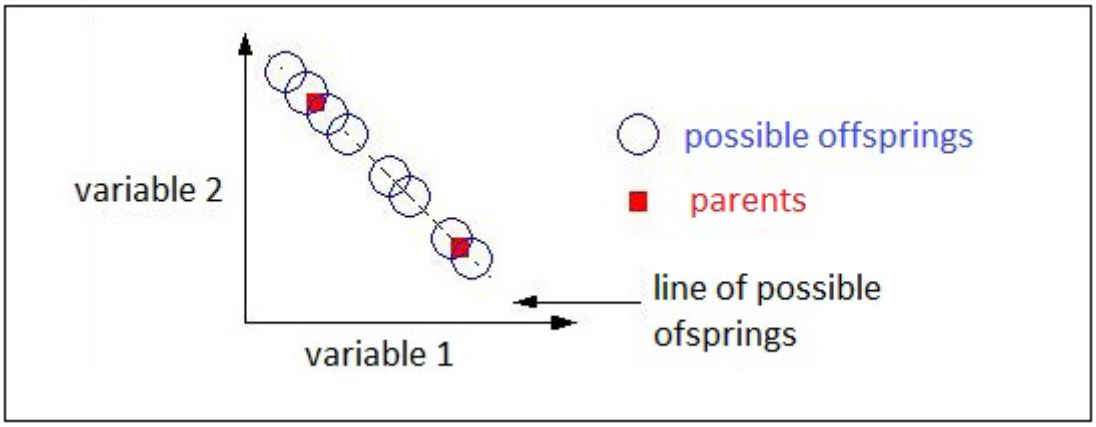



**Figure 45:** Possible positions of the offspring after line recombination (adapted from http://www.geatbx.com/)

### 3.1.3 Binary valued recombination (crossover)

Recombination produces new individuals in combining the information contained in two or more parents (parents – mating population). This is done by combining the variable values of the parents. Depending on the representation of the variables different methods must be used.

During the recombination of binary variables only parts of the individuals are exchanged between the individuals. Depending on the number of parts, the individuals are divided before the exchange of variables (the number of cross points). The number of cross points distinguishes the methods.

**Single-point / double point / multi-point crossover**

In single-point crossover one crossover position $k\_[1,2,\ldots,Nvar\text{-}1]$, *Nvar*: number of variables of an individual, is selected uniformly at random and the variables exchanged between the individuals about this point, then two new offspring are produced. Figure 46 illustrates this process.

*Example*

Consider the following two individuals with 11 binary variables each:

*individual 1* 0 1 1 1 0 0 1 1 0 1 0
*individual 2* 1 0 1 0 1 1 0 0 1 0 1

The chosen crossover positions are:

*crossover position:* 5

After crossover the new individuals are created:

*offspring 1* 0 1 1 1 0| 1 0 0 1 0 1
*offspring 2* 1 0 1 0 1| 0 1 1 0 1 0

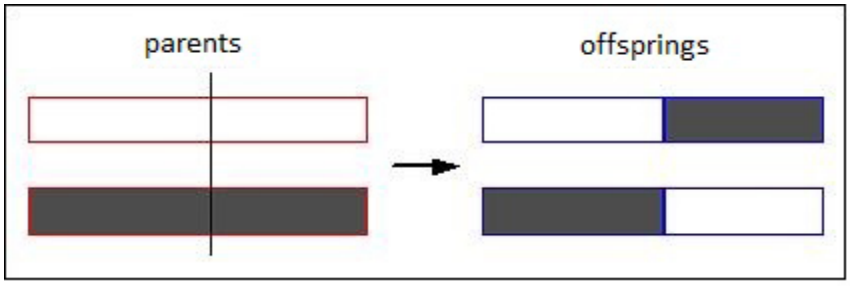



**Figure 46:** Single-point crossover (adapted from http://www.geatbx.com/)

In double-point crossover two crossover positions are selected uniformly at random and the variables exchanged between the individuals between these points, then two new offsprings are produced. Single-point and double-point crossover are special cases of the general method multi-point crossover.

For multi-point crossover, *m* crossover positions $k_i\_[1,2,\ldots,Nvar\text{-}1]$, $i$=1:$m$, *Nvar*: number of variables of an individual are chosen at random with no duplicates and sorted into ascending order. Then, the variables between successive crossover points are exchanged between two parents to produce two new offsprings. The section between the first variable and the first crossover point is not exchanged between individuals. Figure 47 illustrates this process.

*Example*

Consider the following two individuals with 11 binary variables each:

| | |
|---|---|
| *individual 1* | 0 1 1 1 0 0 1 1 0 1 0 |
| *individual 2* | 1 0 1 0 1 1 0 0 1 0 1 |

The chosen crossover positions are:

| | | | |
|---|---|---|---|
| *cross pos. (m=3):* | 2 | 6 | 10 |

After crossover the new individuals are created:

| | |
|---|---|
| *offspring 1* | 0 1\| 1 0 1 1\| 0 1 1 1\| 1 |
| *offspring 2* | 1 0\| 1 1 0 0\| 0 0 1 0\| 0 |

Line recombination can generate any point on the line defined by the parents. Figure 47 shows the possible positions of the offspring after line recombination.

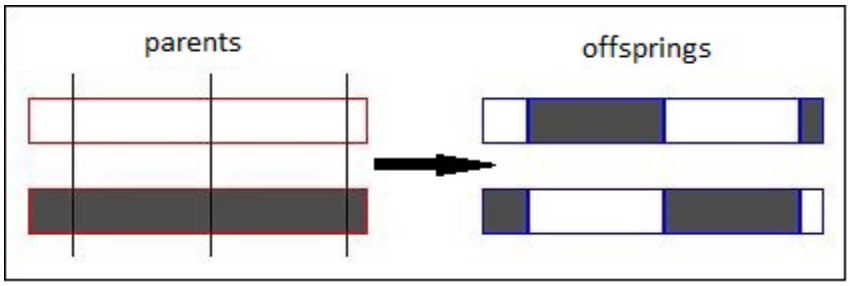



**Figure 47:** Multi-point crossover (adapted from http://www.geatbx.com/)

The idea behind multi-point, and indeed many of the variations on the crossover operator, is that parts of the chromosome representation that contribute most to the performance of a particular individual may not necessarily be contained in adjacent substrings (Booker 1987). Further, the disruptive nature of multi-point crossover appears to encourage the exploration of the search space, rather than favouring the convergence to highly fit individuals early in the search, thus making the search more robust (Spears and De Jong 1991).

**Uniform crossover**

Single and multi-point crossover defines cross points as places between loci where an individual can be split. Uniform crossover (Syswerda 1989) generalizes this scheme to make every locus a potential crossover point. A crossover mask, the same length as the individual structure is created at random and the parity of the bits in the mask indicate which parent will supply the offspring with which bits. This method is identical to discrete recombination.

*Example*

Consider the following two individuals with 11 binary variables each:

| | |
|---|---|
| *individual 1* | 0 1 1 1 0 0 1 1 0 1 0 |
| *individual 2* | 1 0 1 0 1 1 0 0 1 0 1 |

For each variable the parent who contributes its variable to the offspring is chosen randomly with equal probability. Here, the offspring 1 is produced by taking the bit from parent 1 if the corresponding mask bit is 1 or the bit from parent 2 if the corresponding mask bit is 0. Offspring 2 is created using the inverse of the mask, usually.

| | |
|---|---|
| *sample 1* | 0 1 1 0 0 0 1 1 0 1 0 |
| *sample 2* | 1 0 0 1 1 1 0 0 1 0 1 |

After crossover the new individuals are created:

| | |
|---|---|
| *offspring 1* | 1 1 1 0 1 1 1 1 1 1 1 |
| *offspring 2* | 0 0 1 1 0 0 0 0 0 0 0 |

Uniform crossover, like multi-point crossover, has been claimed to reduce the bias associated with the length of the binary representation used and the particular coding for a given parameter set. This helps to overcome the bias in single-point crossover towards short substrings without requiring precise understanding of the significance of the individual bits in the individuals' representation. (Spears and De Jong 1991) demonstrated how uniform crossover may be parametrized by applying a probability to the swapping of bits. This extra parameter can be used to control the amount of disruption during recombination without introducing a bias towards the length of the representation used.

### 3.1.3 Mutation

By mutation individuals are randomly altered. These variations (mutation steps) are mostly small. They will be applied to the variables of the individuals with a low probability (mutation probability or mutation rate). Normally, offspring are mutated after being created by recombination.

**Real valued mutation**

Mutation of real variables means that randomly created values are added to the variables with a low probability. Thus, the probability of mutating a variable (mutation rate) and the size of the changes for each mutated variable (mutation step) must be defined.

The probability of mutating a variable is inversely proportional to the number of variables (dimensions). The more dimensions one individual has, the smaller is the mutation probability. Different papers reported results for the optimal mutation rate. (Mühlenbein and Schlierkamp-Voosen 1993) writes that a mutation rate of $1/n$ ($n$: number of variables of an individual) produced good results for a wide variety of test functions. It means that per mutation only one variable per individual is changed/mutated. Thus, the mutation rate is independent of the size of the population. Similar results are reported in (Bäck 1993) and (Bäck 1996) for a binary valued representation. For unimodal functions a mutation rate of $1/n$ was the best choice. An increase in the mutation rate at the beginning connected with a decrease in the mutation rate to $1/n$ at the end gave only an insignificant acceleration of the search. The given recommendations for the mutation rate are only correct for separable functions. However, most real world functions are not fully separable. For these functions no recommendations for the mutation rate can be given. As long as nothing else is known, a mutation rate of $1/n$ is suggested as well.

The size of the mutation step is usually difficult to choose. The optimal step size depends on the problem considered and may even vary during the optimization process. It is known, that small steps (small mutation steps) are often successful, especially when the individual is already well adapted. However, larger changes (large mutation steps) can produce good results much quicker. Thus, a good mutation operator should often produce small step sizes with a high probability and large step sizes with a low probability.

In (Mühlenbein and Schlierkamp-Voosen 1993) and (Mühlenbein 1994) such an operator is proposed (mutation operator of the Breeder Genetic Algorithm):

$$
\begin{aligned}
&Var_i^{Mut} = Var_i + s_i \cdot r_i a_i \quad i \in \{1,2,\cdots,n\} \text{ uniform at random,}\\
&s_i \in \{-1,1\} \text{ uniform at random,}\\
&r_i = r \cdot domain_i,\ r: \text{mutation range}(\text{standard}: 10\%),\\
&a_i 2^{-u \cdot k}, u \in [0,1] \text{ uniform at random, } k: \text{mutation precision.}
\end{aligned}
$$

This mutation algorithm is able to generate most points in the hypercube defined by the variables of the individual and range of the mutation (the range of mutation is given by the value of the parameter *r* and the domain of the variables). Most mutated individuals will be generated near the individual before mutation. Only some mutated individuals will be far away from the not mutated individual. That means, the probability of small step-sizes is greater than that the probability of bigger steps. Figure 48 tries to give an impression of the mutation results of this mutation operator.

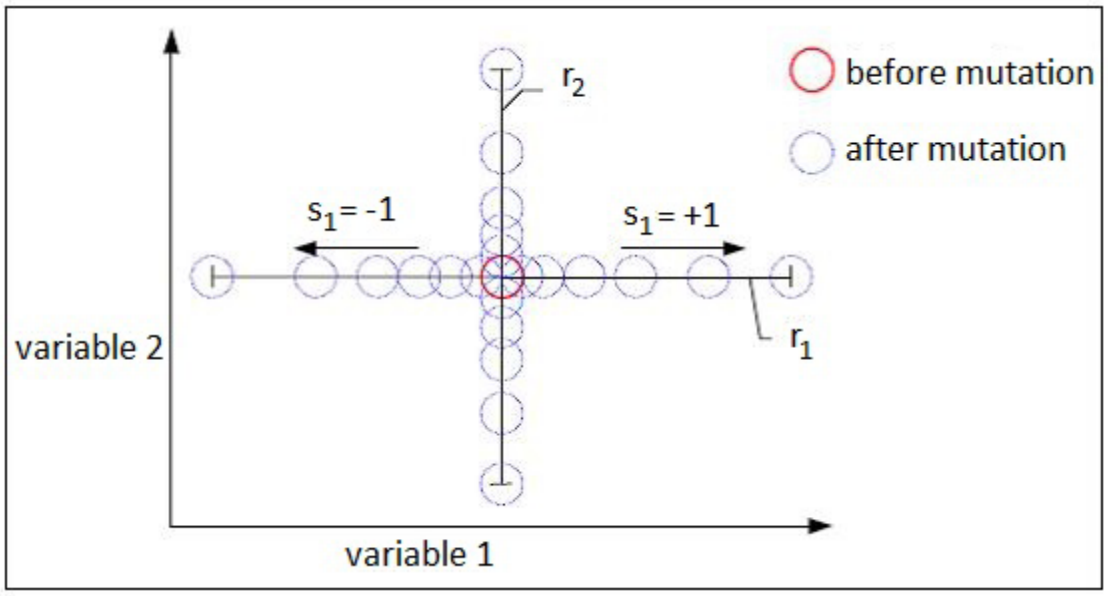



**Figure 48:** Effect of mutation of real variables in two dimensions (adapted from http://www.geatbx.com/)

The parameter *k* (mutation precision) defines indirectly the minimal step size possible and the distribution of mutation steps inside the mutation range. The smallest relative mutation step size is $2^{-k}$, the largest $2^0 = 1$. Thus, the mutation steps are created inside the area $[r, r \cdot 2^{-k}]$ (*r*: mutation range). With a mutation precision of $k = 16$, the smallest mutation step possible is $r \cdot 2^{-16}$. Thus, when the variables of an individual are so close to the optimum, a further improvement is not possible. This can be circumvented by decreasing the mutation range (restart of the evolutionary run or use of multiple strategies)

Typical values for the parameters of the mutation operator are the following:

mutation precision *k*: $k \in \{4, 5, \ldots, 20\}$
mutation range *r*: $r \in [0.1, 10^{-6}]$

By changing these parameters, very different search strategies can be defined.

**Binary mutation**

For binary valued individuals mutation means the flipping of variable values, because every variable has only two states. Thus, the size of the mutation step is always 1. For every individual the variable value to change is chosen (mostly uniform at random). Figure 49 shows an example of a binary mutation for an individual with 11 variables, where variable 4 is mutated.

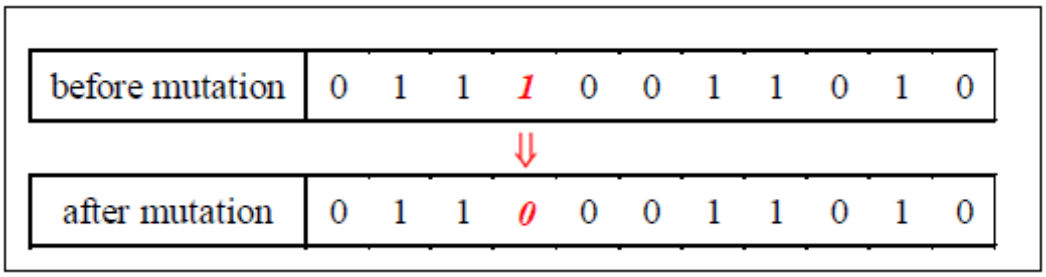



**Figure 49:** Individual before and after binary mutation (adapted from http://www.geatbx.com/)

Assuming that the above individual decodes a real number in the bounds [1, 10], the effect of the mutation depends on the actual coding. Figure 50 shows the different numbers of the individual before and after mutation for binary/grey and arithmetic/logarithmic coding.

| scaling | linear | | logarithmic | |
|---|---|---|---|---|
| coding | binary | gray | binary | gray |
| before mutation | 5.0537 | 4.2887 | 2.8211 | 2.3196 |
| after mutation | 4.4910 | 3.3346 | 2.4428 | 1.8172 |

**Figure 50:** Result of the binary mutation (adapted from http://www.geatbx.com/)

However, there is no longer a reason to decode real variables into binary variables. The advantages of mutation operators for real variables were shown in some publications, e.g. (Michalewicz 1994) and (Davis 1991).

## 3.2 Genetic algorithms

A genetic algorithm is a type of a searching algorithm. It searches a solution space for an optimal solution to a problem. The key characteristic of the genetic algorithm is how the searching is done. The algorithm creates a "population" of possible solutions to the problem and lets them "evolve" over multiple generations to find better and better solutions. The generic form of the genetic algorithm is shown in Figure 51. The items in bold in the algorithm are defined here.

1. Create a **population** of random candidate solutions named *pop*.
2. Until the algorithm termination conditions are met, do the following (each iteration is called a generation):
   a) Create an empty population named *new-pop*.
   b) While *new-pop* is not full, do the following:
      1) **Select** two **individuals** at random from *pop* so that individuals which are more **fit** are more likely to be selected.
      2) **Cross-over** the two individuals to produce two new individuals.
   c) Let each individual in *new-pop* have a random chance to **mutate**.
   d) Replace *pop* with *new-pop*.
3. Select the individual from *pop* with the highest **fitness** as the solution to the problem.

**Figure 51:** The Genetic Algorithm

The **population** consists of the collection of candidate solutions that we are considering during the course of the algorithm. Over the generations of the algorithm, new members are "born" into the population, while others "die" out of the population. A single solution in the population is referred to as an **individual**. The **fitness** of an individual is a measure of how "good" is the solution represented by the individual. The better solution has a higher fitness value – obviously, this is dependent on the problem to be solved. The **selection** process is analogous to the survival of the fittest in the natural world. Individuals are selected for "breeding" (or **cross-over**) based upon their fitness values. The crossover occurs by mingling two solutions together to produce two new individuals. During each generation, there is a small chance for each individual to **mutate**.

To use a genetic algorithm, there are several questions that need to be answered:

- How is an individual represented?
- How is an individual's fitness calculated?
- How are individuals selected for breeding?
- How are individuals crossed-over?
- How are individuals mutated?
- What is the size of the population?
- What are the "termination conditions"?

Most of these questions have problem specific answers. The last two, however, can be discussed in a more general way.

The size of the population is highly variable. The population should be as large as possible. The limiting factor is, of course, the running time of the algorithm. The larger population means more time consuming calculation.

The algorithm in Figure 51 has a very vague end point – the meaning of "until the termination conditions are met" is not immediately obvious. The reason for this is that there is no one way to end the algorithm. The simplest approach is to run the search for a set number of generations – the longer. Another approach is to end the algorithm after a certain number of generations pass with no improvement of the fitness of the best individual in the population. There are other possibilities as well. Since most of the other questions are dependent upon the search problem, we will look at two example problems that can be solved using genetic algorithms: finding a mathematical function's maximum and the travelling salesman problem.

### 3.2.1 Function maximization

*Example* (Thede 2004) One application for a genetic algorithm is to find values for a collection of variables that will maximize a particular function of those variables. While this type of problem could be solved otherwise, it is useful as an example of the operation of genetic algorithms. For this example, let's assume that we are trying to determine such variables that produce the maximum value for this function:

$$f(w, x, y, z) = w^3 + x^2 - y^2 - z^2 + 2yz - 3wx + wz - xy + 2$$

This could probably be solved using multivariable calculus, but it is a good simple example of the use of genetic algorithms. To use the genetic algorithm, we need to answer the questions listed in the previous section.

**How is an individual represented?**

What information is needed to have a "solution" of the maximization problem? It is clear that we need only values: *w, x, y*, and *z*. Assuming that we have values for these four variables, we have a candidate solution for our problem.

The question is how to represent these four values. A simple way to do this is to use an array of four values (integers or floating point numbers). However, for genetic algorithms it is usually better to have a larger individual – this way, variations can be done in a more subtle way. The research shows (Holland 1975) that representation of individuals using bit strings offers the best performance. We can simply choose a size in bits for each variable, and then concatenate the four values together into a single bit string.

For example, we will choose to represent each variable as a four-bit integer, making our entire individual a 16-bit string. Thus, an individual such as

*1101 0110 0111 1100*

represents a solution where $w = 13$, $x = 6$, $y = 7$, and $z = 12$.

**How is an individual's fitness calculated?**

Next, we consider how to determine the fitness of each individual. There is generally a differentiation between the *fitness* and *evaluation* functions. The evaluation function is a function that returns an absolute measure of the individual. The fitness function is a function that measures the value of the individual relative to the rest of the population.

In our example, an obvious evaluation function would be to simply calculate the value of $f$ for the given variables. For example, assume we have a population of 4 individuals:

*1010 1110 1000 0011*
*0110 1001 1111 0110*
*0111 0110 1110 1011*
*0001 0110 1000 0000*

The first individual represents $w = 10$, $x = 14$, $y = 8$, and $z = 3$, for an $f$ value of 671. The values for the entire population can be seen in the following table:

| ***Individual*** | ***w*** | ***x*** | ***y*** | ***z*** | ***f*** |
|---|---|---|---|---|---|
| 1010111010000011 | 10 | 14 | 8 | 3 | 671 |
| 0110100111110110 | 6 | 9 | 15 | 6 | -43 |
| 0111011011101011 | 7 | 6 | 14 | 11 | 239 |
| 0001011010000000 | 1 | 6 | 8 | 0 | -91 |

The fitness function can be chosen from many options. For example, the individuals could be listed in order from the lowest to the highest evaluation function values, and an ordinal ranking applied. OR The fitness function could be the individual's evaluation value divided by the average evaluation value. Looking at both of these approaches would give us something like this:

| *Individual* | *evaluation* | *ordinal* | *averaging* |
|---|---|---|---|
| 1010111010000011 | 671 | 4 | 2.62 |
| 0110100111110110 | -43 | 2 | 0.19 |
| 0111011011101011 | 239 | 3 | 0.81 |
| 0001011010000000 | -91 | 1 | 0.03 |

The key is that the fitness of an individual should represent the value of the individual relative to the rest of the population, so that the best individual has the highest fitness.

**How are individuals selected for breeding?**

The key to the selection process is that it should be probabilistically weighted so that higher fitness individuals have a higher probability of being selected. Other than these specifications, the method of selection is open to interpretation.

One possibility is to use the ordinal method for the fitness function, then calculate a probability of selection that is equal to the individual's fitness value divided by the total fitness of all the individuals. In the example above, that would give the first individual a 40% chance of being selected, the second a 20% chance, the third a 30% chance, and the fourth a 10% chance. It gives better individuals more chances to be selected.

A similar approach could be used with the average fitness calculation. This would give the first individual a 72% chance, the second a 5% chance, the third a 22% chance, and the fourth a 1% chance. This method makes the probability more dependent on the relative evaluation functions of each individual.

**How are individuals crossed-over?**

Once we have selected a pair of individuals, they are "bred" – or in genetic algorithm language, they are *crossed-over*. Typically two children are created from each set of parents. One method for performing the cross-over is described here, but there are other approaches. Two locations are randomly chosen within the individual. These define corresponding substrings in each individual. The substrings are swapped between two parent individuals, creating two new children. For example, let's look at our four individuals again:

*1010 1110 1000 0011*
*0110 1001 1111 0110*
*0111 0110 1110 1011*
*0001 0110 1000 0000*

Let's assume that the first and third individuals are chosen for cross-over. Keep in mind that the selection process is random. The fourth and fourteenth bits are randomly selected to define the substring to be swapped, so the cross-over looks like this:

*1010111010000011* *101**0111010000**011* *1011011011101011*
→ →
*0111011011101011* *011**1011011101**011* *0110111010000011*

Thus, two new individuals are created. We should create new individuals until we replace the entire population – in our example, we need one more cross-over operators. Assume that the first and fourth individuals are selected this time. Note that an individual may be selected multiple times for breeding, while other individuals might never be selected. Further assume that the eleventh and sixteenth bits are randomly selected for the cross-over point. We could apply the second cross-over like this:

*1010111010000011* *1010111010000011* *1010111010000000*
→ →
*0001011010000000* *0001011010000000* *0001011010000011*

The second generation of the population is the following:

*1011 0110 1110 1011*
*0110 1110 1000 0011*
*1010 1110 1000 0000*
*0001 0110 1000 0011*

**How are individuals mutated?**

Finally, we need to allow individuals to mutate. When using bit strings, the easiest way to implement the mutation is to allow every single bit in every individual a chance to mutate. This chance should be very small, since we don't want to have individuals changing dramatically due to mutation. Setting the percentage so, that roughly one bit per individual has a chance to change on average.

The mutation will consist of having the bit "flip": *1* changes to *0* and *0* changes to *1*. In our example, assume that the bold and italicized bits have been chosen for mutation:

*1011011011101011* → *1011011011101011*
*01101**1**1010000011* → *01101**0**1010000011*
*10101110100**0**0000* → *10101110100**1**0000*
*0**0**01011010000011* → *0**1**01011010000000**1***

**Wrapping Up**

Finally, let's look at the next population:

| *Individual* | *w* | *x* | *y* | *z* | *f* |
|---|---|---|---|---|---|
| 1011011011101011 | 11 | 6 | 14 | 11 | 1,045 |
| 0110101010000011 | 6 | 10 | 8 | 3 | 51 |
| 1010111010010000 | 10 | 14 | 9 | 0 | 571 |
| 0101011010000001 | 5 | 6 | 8 | 1 | -19 |

The average evaluation value is 412, versus an average of 194 for the previous generation. Clearly, this is a constructed example, but the exciting thing about genetic algorithms is that this sort of improvement actually does occur in practice.

### 3.2.2 The travelling salesman problem

*Example* (Thede 2004)

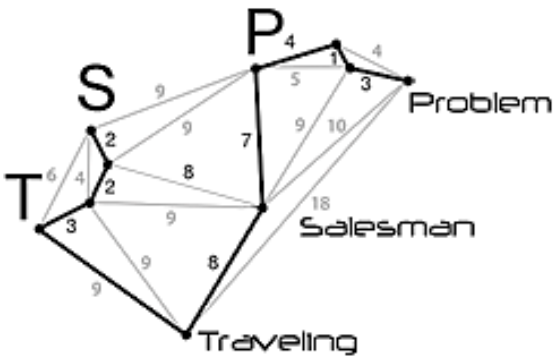



**Figure 52:** Traveling Salesman Problem (adapted from http://www.amdusers.com/wiki/tiki-index.php?page=TSP/)

Now let's look at a less contrived example. Genetic algorithms can be used to solve the traveling salesman problem (TSP), see Figure 52. For those who are unfamiliar with this problem, it can be stated in two ways. Informally, there is a traveling salesman who services some number of cities, including his home city. He needs to travel on a trip such that he starts in his home city, visits every other city exactly once, and returns home. He wants to set up the trip so that it costs him the least amount of money possible. The more formal way of stating the problem casts it as a graph problem. Given a weighted graph with $N$ vertices, find the lowest cost path from some city $v$ that visits every other node exactly once and returns to $v$. For a more thorough discussion of TSP, see (Garey and Johnson 1979).

The problem with TSP is that it is an NP-complete problem. The only known way to find the answer is to list every possible route and find the one with the lowest cost. Since there are a total of $(N - 1)!$ routes, this quickly becomes intractable for large $N$. There are approximation algorithms that run in a reasonable time and produce reasonable results – a genetic algorithm is one of them.

**Individual Representation and Fitness**

Our first step is to decide on a representation for an individual candidate solution, or *tour*. The bit string model is not very useful because cross-overs and mutations should produce a tour that is invalid. Remember that every city has to occur in the tour exactly once except the home city.

The only real choice for representing the individual is a vector or array of cities, most likely stored as integers. The costs of travel between cities should be provided. Using a vector of integers causes some problems with cross-over and mutation, as we'll see in the next section.

For example, the following file defines a TSP with four cities:

*0 2 6 3*
*2 0 9 7*
*6 9 0 8*
*3 7 8 0*

This file shows that the cost of travel from city 0 to city 2 is 6, while the cost from city 3 to city 1 is 7. Then we could represent an individual as a vector of five cities:

[*0 2 1 3 0*]

We also need to be careful when calculating fitness. The clear choice for the evaluation function is the cost of the tour. Remember that a good tour is one whose cost is low, so we need to calculate fitness so that a low cost tour – it corresponds to high fitness individual (path cost).

**Cross-over and Mutation**

When performing cross-over and mutation, we need to make sure that we produce valid tours. This means modifying the cross-over and mutation process. We can still use the same basic idea, however, choose a substring of the vector of cities at random for cross-over, and choose a single point in the vector at random for mutation. The mechanics are a bit different.

For cross-over, rather than simply swapping the substrings (which could easily result in an invalid tour), we will instead keep the same cities, but change their order to match the other parent of the cross-over. For example, assume we have the following two individuals, with the third and sixth cities chosen for the cross-over substring

*7 3* ***6 1 0 2*** *5 4*
*5 4* ***1 7 2 3*** *6 0*

Rather than swapping the cities, which would result in duplications of cities within each tour, we keep the cities in each tour the same, but we reorder the cities to match their order in the other parent. In the above example, we would replace the "**6 1 0 2**" section in the first parent with "**1 2 6 0**", because that is the order in which those four cities appear in the second parent. Similarly, the "**1 7 2 3**" section in the second parent is replaced with "**7 3 1 2**", and we get offspring

*7 3* ***1 2 6 0*** *5 4*
*5 4* ***7 3 1 2*** *6 0*

This allows the concept of the cross-over to remain – that each parent contributes to the construction of new individuals – while guaranteeing that a valid tour is created.

There are a number of approaches for mutation. The simplest is that whenever a city is chosen as the location of a mutation, it is simply swapped with the next city in the tour. For example, if the 6 is chosen for mutation in this tour:

*7 3 1 2* ***6 0*** *5 4*

we would get this tour after the mutation occurs:

*7 3 1 2* ***0 6*** *5 4*

Other options for mutation include selecting two cities at random and swapping them, or selecting an entire substring at random and reversing it, but the concept remains the same – making a relatively small change to an individual.

## 3.3 Genetic programming

Genetic programming is much more powerful than genetic algorithms. The output of the genetic algorithm is a quantity, while the output of the genetic programming is a computer program. In essence, this is the beginning of computer programs that program themselves. Genetic programming works best for several types of problems. The first type is where there is no ideal solution, (for example, a program that drives a car). There is no one solution to driving a car. Some solutions drive safely at the expense of time, while others drive fast at a high safety risk. Therefore, driving a car consists of making compromises of speed versus safety, as well as many other variables. In this case genetic programming will find a solution that attempts to compromise and be the most efficient solution from a large list of variables. Furthermore, genetic programming is useful in finding solutions where the variables are constantly changing. In the previous car example, the program will find one solution for a smooth concrete highway, while it will find a totally different solution for a rough unpaved road.

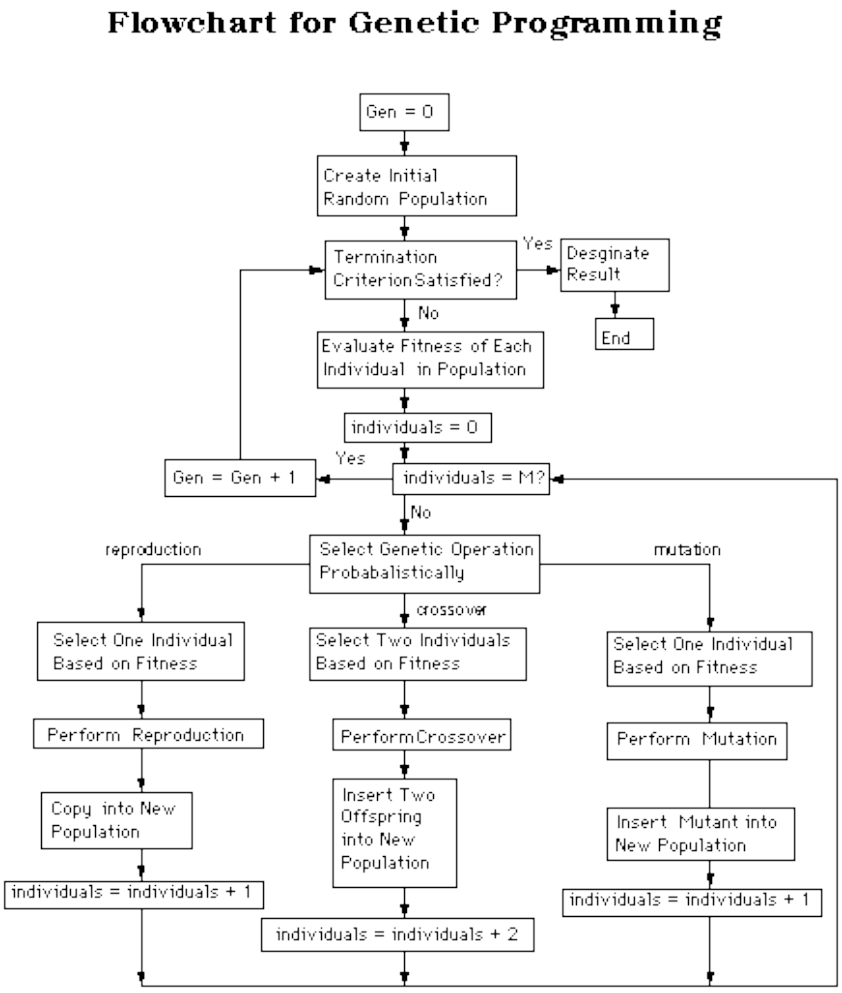



**Figure 53:** Result of the binary mutation (adapted from http://www.geneticprogramming.com)

The main difference between genetic programming and genetic algorithms is the representation of the solution. Genetic programming creates computer programs in the scheme computer languages as the solution. Genetic algorithms create a string of numbers that represent the solution. Genetic programming uses four steps to solve problems:

1. Generate an initial population of random compositions of the functions and terminals of the problem (computer programs).
2. Execute each program in the population and assign it a fitness value according to how well it solves the problem.
3. Create a new population of computer programs.
   1) Copy the best existing programs
   2) Create new computer programs by mutation.
   3) Create new computer programs by crossover (sexual reproduction).
4. The best computer program that appeared in any generation, the best-so-far solution, is designated as the result of genetic programming (Koza 1992).

The flowchart for Genetic Programming (GP) is shown in Figure 53.

The most difficult and most important concept of genetic programming is the *fitness function*. The fitness function determines how well a program is able to solve the problem. It varies greatly from one type of program to the next. For example, if one were to create a genetic program to set the time of a clock, the fitness function would simply be the amount of time that the clock is wrong. Unfortunately, few problems have such an easy fitness function; most cases require a slight modification of the problem in order to find the fitness.

The terminal and function sets are also important components of genetic programming. The terminal and function sets are the alphabet of the programs to be made. The terminal set consists of the variables and constants of the programs. Functions are several mathematical functions, such as addition, subtraction, division, multiplication and other more complex functions.

### 3.3.1 Operators of genetic programming

**Crossover Operator**

Two primary operations exist for modifying structures in genetic programming. The most important one is the crossover operation. In the crossover operation, two solutions are combined to form two new solutions or offspring. The parents are chosen from the population by a function of the fitness of the solutions. Three methods exist for selecting the solutions for the crossover operation.

The first method uses probability based on the fitness of the solution. If $f(s_i(t))$ is the fitness of the solution $S_i$ and

$$F = \sum_{i=1}^{M} f(s_i(t))$$

$F$ is the total sum of all the members of the population $M$. The probability to solution $S_i$ is copied to the next generation is (Koza 1992) then:

$$p_i = \frac{f(s_i(t))}{\sum_{i=1}^{M} f(s_i(t))}$$

Another method for selecting the solution to be copied is tournament selection. Typically the genetic program chooses two solutions random. The solution with the higher fitness will win. This method simulates biological mating patterns in which, two members of the same sex compete to mate with a third one of a different sex. Finally, the third method is done by rank. In rank selection, selection is based on the rank, (not the numerical value) of the fitness values of the solutions of the population (Koza 1992).

The creation of offsprings from the crossover operation is accomplished by deleting the crossover fragment of the first parent and then inserting the crossover fragment of the second parent. The second offspring is produced in a symmetric manner. For example consider the two *S*-expressions in Figure 54, written in a modified scheme programming language and represented in a tree.

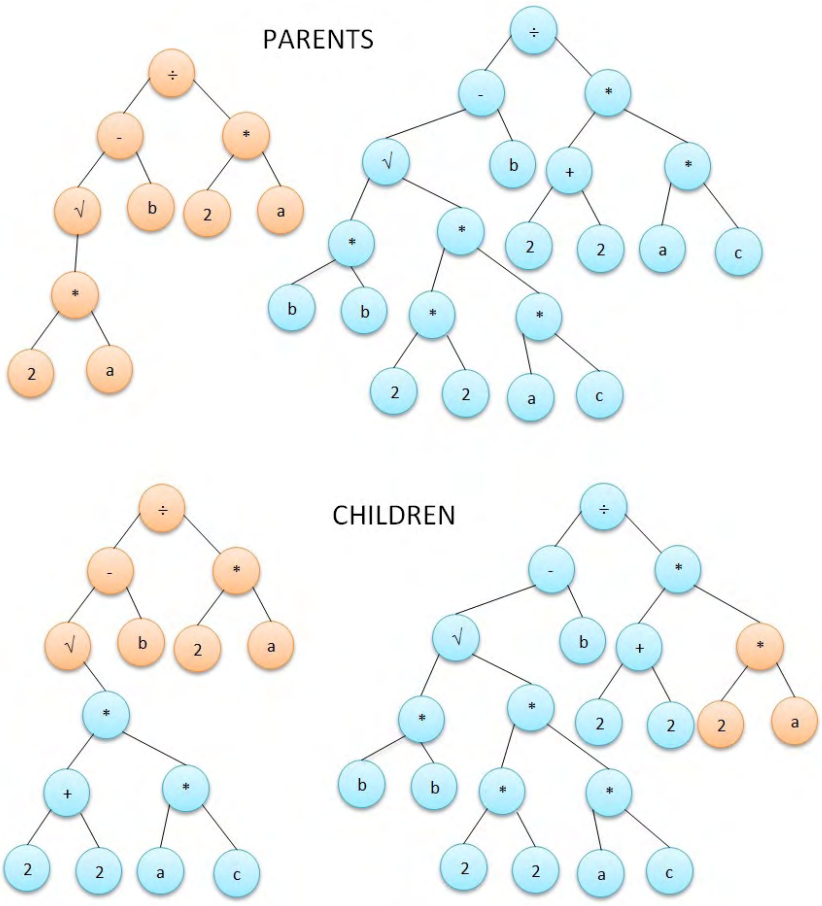



**Figure 54:** Crossover operation for genetic programming. The bold selections on both parents are swapped to create the offspring or children. The child on the right is the parse tree representation for the quadratic equation. (adapted from http://www.geneticprogramming.com)

An important improvement that genetic programming displays over genetic algorithms is its ability to create two new solutions from the same solution. In Figure 55 the same parent is used twice to create two new children. This figure illustrates one of the main advantages of genetic programming over genetic algorithms. In genetic programming identical parents can yield different offspring, while in genetic algorithms identical parents would yield identical offspring. The bold selections indicate the subtrees to be swapped.

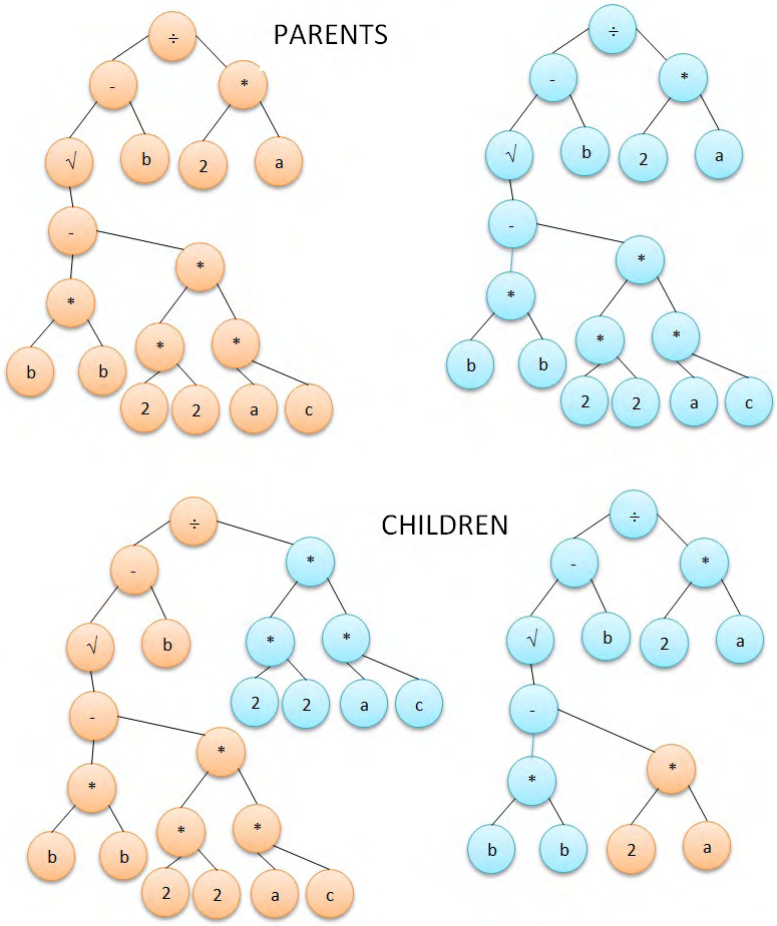



**Figure 55:** Crossover operation for identical parents. (adapted from http://www.geneticprogramming.com)

### Mutation Operator

Mutation is another important feature of genetic programming. Two types of mutations are possible. In the first kind a function can only replace a function or a terminal can only replace a terminal. In the second kind an entire subtree can replace another subtree. Figure 56 explains the concept of mutation. Genetic programming uses two different types of mutations. The top parse tree is the original agent. The bottom left parse tree illustrates a mutation of a single terminal (2) for another single terminal (a). It also illustrates a mutation of a single function (-) for another single function (+). The parse tree on the bottom right illustrates a replacement of a subtree by another subtree.

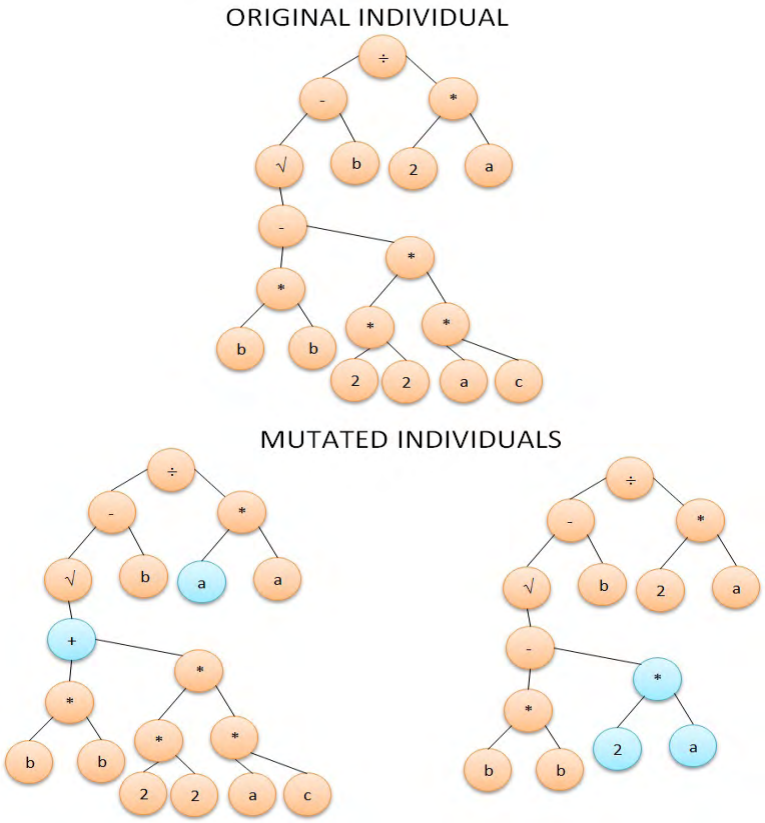



**Figure 56:** Mutation operation. (adapted from http://www.geneticprogramming.com)

### 3.3.2 Applications of genetic programming

Genetic programming can be used for example in the following task solving:

**Gun Firing Program.** A more complicated example consists of training a genetic program to fire a gun to hit a moving target. The fitness function is the distance that the bullet is off from the target. The program has to learn to take into account a number of variables, such as the wind velocity, the type of gun used, the distance to the target, the height of the target, the velocity and acceleration of the target. This problem represents the type of problem for which genetic programs are best. It is a simple fitness function with a large number of variables.

**Water Sprinkler System.** Consider a program to control the flow of water through a system of water sprinklers. The fitness function is the correct amount of water evenly distributed over the surface. Unfortunately, there is no one variable encompassing this measurement. Thus, the problem must be modified to find a numerical fitness. One possible solution is placing water-collecting measuring devices at certain intervals on the surface. The fitness could then be the standard deviation in water level from all the measuring devices. Another possible fitness measure could be the difference between the lowest measured water level and the ideal amount of water; however, this number would not account in any way the water marks at other measuring devices, which may not be at the ideal mark.

**Maze Solving Program.** If one were to create a program to find the solution to a maze, first, the program would have to be trained with several known mazes. The ideal solution from the start to the finish of the maze would be described by a path of dots. The fitness in this case would be the number of dots the program is able to find. In order to prevent the program from wandering around the maze too long, a time limit is implemented along with the fitness function.

## 3.4 Differential evolution

Differential Evolution (DE) is a population-based optimization method that works on real-number-coded individuals (Storn and Price 1997). DE is quite robust, fast, and effective, with global optimization ability. It does not require the objective function to be differentiable, and it works well even with noisy and time-dependent objective functions. Differential Evolution is a parallel direct search method which utilizes *NP* parameter vectors $x_{i,G}$, $i = 0, 1, 2, \ldots, NP\text{-}1$ as a population for each generation *G*. *NP* does not change during the minimization process. The initial population is chosen randomly if nothing is known about the system. As a rule, we will assume a uniform probability distribution for all random decisions unless otherwise stated. In case a preliminary solution is available, the initial population is often generated by adding normally distributed random deviations to the nominal solution $x_{nom,0}$. The crucial idea behind DE is a scheme for generating trial parameter vectors. DE generates new parameter vectors by adding a weighted difference vector between two population members to a third member. If the resulting vector yields a lower objective function value than a predetermined population member, the newly generated vector will replace the vector with which it was compared in the following generation. The comparison vector can but need not be part of the generation process mentioned above. In addition the best parameter vector $x_{best,G}$ is evaluated for every generation *G* in order to keep track of the progress that is made during the minimization process.

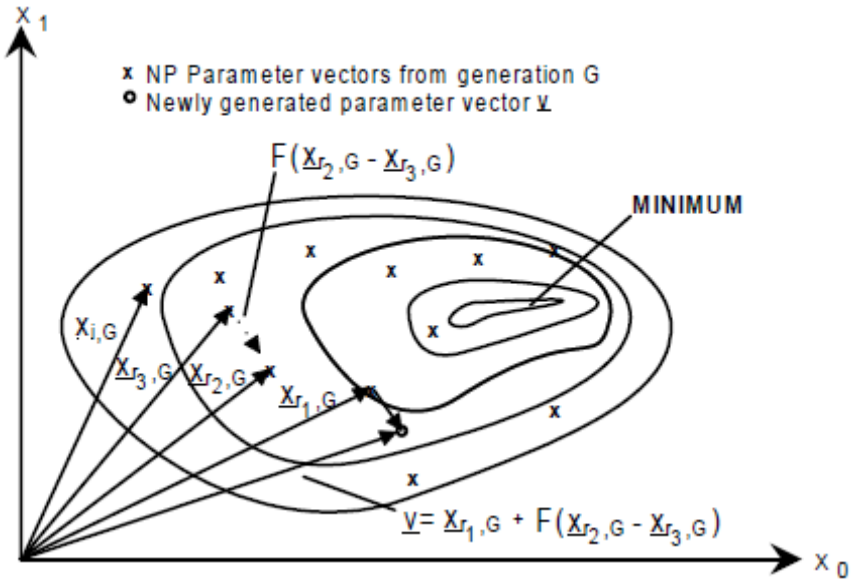



**Figure 57:** Two-dimensional example of an objective function showing its contour lines and the process for generating v in scheme DE1. The weighted difference vector of two arbitrarily chosen vectors is added to a third vector to yield the vector *v*.

It is extracting distance and direction information from the population to generate random deviations results in an adaptive scheme with excellent convergence properties. Several variants of DE have been tried, the two most promising of which are subsequently presented in greater detail.

**Scheme DE1**

The first variant of DE works as follows: for each vector $x_{i,G}$, $i$ = 0, 1, 2, … , $NP$-1, a trial vector $v$ is generated according to

$$v = x_{r_1,G} + F \cdot \left(x_{r_2,G} - x_{r_3,G}\right),$$

with $r_1$, $r_2$, $r_3 \in$ [0, $NP$-1], integer and mutually different, and $F > 0$. Integers $r_1$, $r_2$ and $r_3$ are chosen randomly from the interval [0, $NP$-1] and are different from the running index $i$. $F$ is a real and constant factor which controls the amplification of the differential variation $\left(x_{r_2,G} - x_{r_3,G}\right)$. Figure 57 (Storn and Price 1997) shows a two-dimensional example that illustrates the different vectors that are used in DE1.

In order to increase the diversity of the parameter vectors, the vector

$$u = (u_0, u_1, \cdots, u_{D-1})^T$$

with $u_j = \begin{cases} v_j & \text{for } j = \langle n \rangle_D, \langle n+1 \rangle_D, \cdots, \langle n+L-1 \rangle_D \\ (x_{i,G})_j & \text{for all other } j \in [0, D-1] \end{cases}$

is formed where the acute brackets $\langle\ \rangle_D$ denote the modulo function with modulus $D$.

Equations yield a certain sequence of the vector elements of $u$ to be identical to the elements of $v$, the other elements of $u$ acquire the original values of $x_{i,G}$. Choosing a subgroup of parameters for mutation is similar to a process known as crossover in Genetic Algorithms. This idea is illustrated in Figure 58 (Storn and Price 1997) for $D = 7$, $n = 2$ and $L = 3$. The starting index $n$ is a randomly chosen integer from the interval [0, $D$-1]. The integer $L$, which denotes the number of parameters that are going to be exchanged, is drawn from the interval [1, $D$]. The algorithm which determines L works according to the following lines of pseudo code where *rand*( ) is supposed to generate a random number ∈ [0,1):

```
L = 0;
do {
L = L + 1;
}while(rand()< CR) AND (L < D));
```

Hence the probability $\Pr(L >= \nu) = (CR)^{\nu-1}$, $\nu > 0$. $CR \in [0,1]$ is the crossover probability and constitutes a control variable for the DE1-scheme. The random decisions for both $n$ and $L$ are made anew for each trial vector $v$.

In order to decide whether the new vector u shall become a population member of generation $G$+1, it will be compared to $x_{i,G}$. If vector $u$ yields a smaller objective function value, than $x_{i,G}$, $x_{i,G+1}$ is set to $u$, otherwise the old value $x_{i,G}$ is retained.

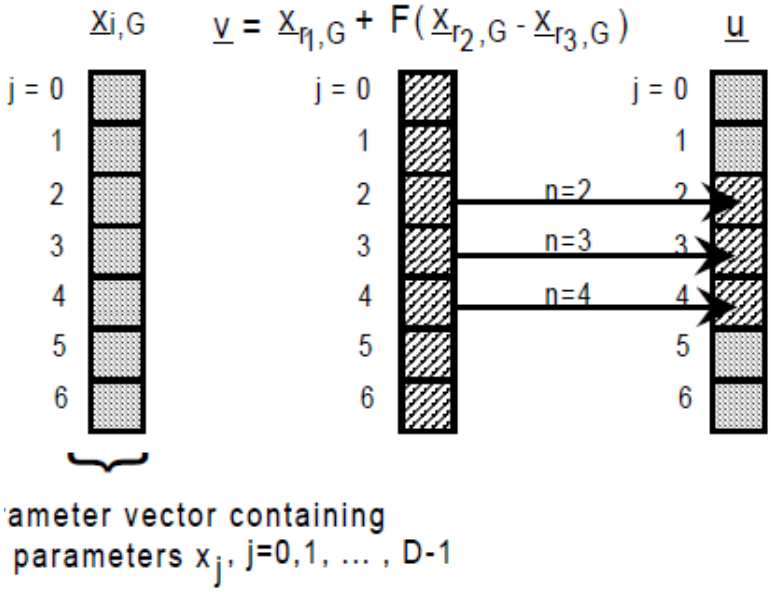



**Figure 58:** Illustration of the crossover process for *D*=7, *n*=2 and *L*=3.

**Scheme DE2**

Basically, scheme DE2 works in the same way as DE1 but generates the vector v according to

$$v = x_{r_1,G} + \lambda \cdot \left(x_{best,G} - x_{i,G}\right) + F \cdot \left(x_{r_2,G} - x_{r_3,G}\right),$$

introducing an additional control variable $\lambda$. The idea behind $\lambda$ is to provide a means to enhance the greediness of the scheme by incorporating the current best vector $x_{best,G}$. This feature can be useful for objective functions where the global minimum is relatively easy to find. Figure 59 (Storn and Price 1997) illustrates the vector-generation process defined by the previous equation. The construction of $u$ from $v$ and $x_{i,G}$ as well as the decision process are identical to DE1.

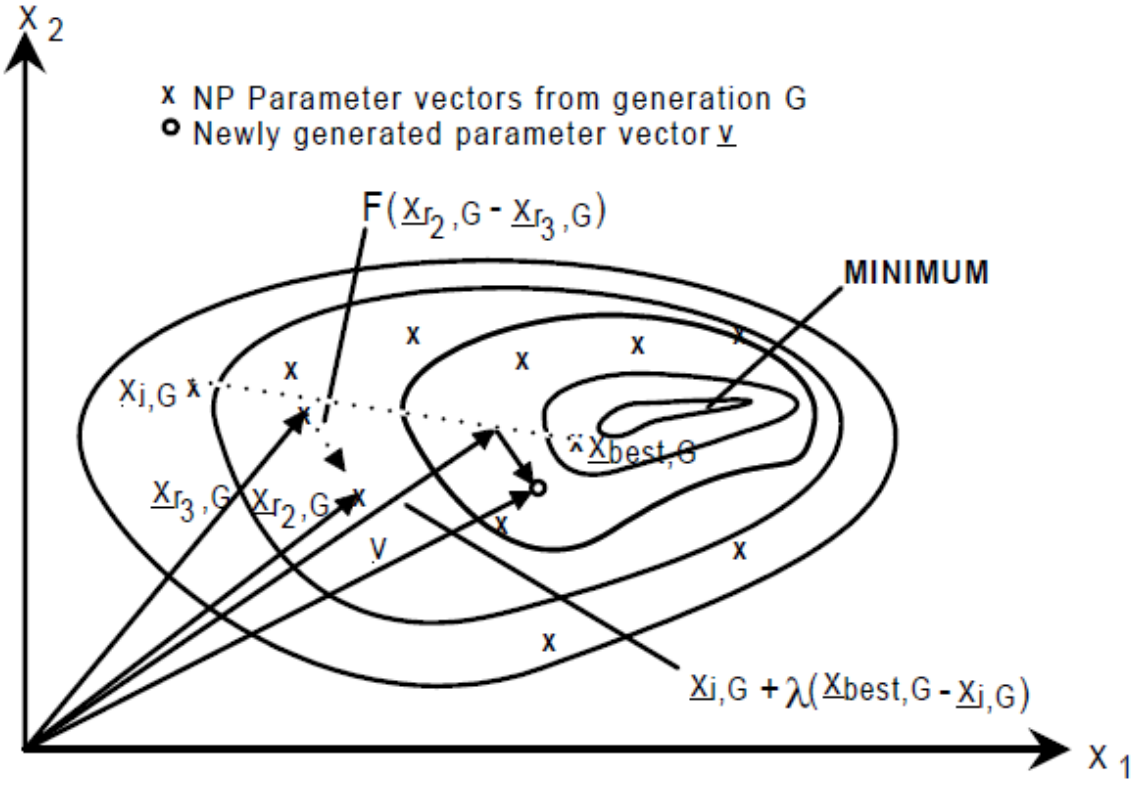



**Figure 59:** Two dimensional example of an objective function showing its contour lines and the process for generating v in scheme DE2.

### Canonical DE

A schematic of the canonical DE strategy is given in Figure 60 (Price 1999). There are essentially five sections to the code depicted in figure 60 (Price 1999).

Section *1* describes the input to the heuristic. *D* is the size of the problem, *Gmax* is the maximum number of generations, *NP* is the total number of solutions, *F* is the scaling factor of the solution and *CR* is the factor for crossover. *F* and *CR* together make the internal tuning parameters for the heuristic.

Section *2* in figure 60 outlines the initialization of the heuristic. Each solution $x_{i,j,G}$=0 is created randomly between the two bounds $x^{(lo)}$ and $x^{(hi)}$. The parameter *j* represents the index to the values within the solution and parameter *i* indexes the solutions within the population. So, to illustrate, $x_{4,2,0}$ represents the fourth value of the second solution at the initial generation.

After initialization, the population is subjected to repeated iterations in section *3*.

Section *4* describes the conversion routines of DE. Initially, three random numbers $r_1$, $r_2$, $r_3$ are selected, unique to each other and to the current indexed solution $i$ in the population in 4.1. Henceforth, a new index $j_{rand}$ is selected in the solution, $j_{rand}$ points to the value being modified in the solution as given in 4.2. In 4.3, two solutions, $x_{j,r1,G}$ and $x_{j,r2,G}$ are selected through the index $r_1$ and $r_2$ and their values subtracted. This value is then multiplied by $F$, the predefined scaling factor. This is added to the value indexed by $r_3$. However, this solution is not arbitrarily accepted in the solution. A new random number is generated, and if this random number is less than the value of $CR$, then the new value replaces the old value in the current solution. The fitness of the resulting solution, referred to as a perturbed vector $u_{j,i,G}$, is then compared with the fitness of $x_{j,i,G}$. If the fitness of $u_{j,i,G}$ is greater than the fitness of $x_{j,i,G}$, then $x_{j,i,G}$ is replaced with $u_{j,i,G}$; otherwise, $x_{j,i,G}$ remains in the population as $x_{j,i,G+1}$. Hence, the competition is only between the new *child* solution and its *parent* solution.

The description of some of the commonly used basic DE strategies is presented in Table 4. The description of all 10 basic strategies is described in (Price 1999) or (Storn. and Price 1997). These strategies differ in the way of calculating the perturbed vector $u_{j,i,G}$. Scaling vector $F$ is replaced with a randomly generated vector $F_{Rand}$ in *DELocalToBest* strategy and with a randomly generated vector $F_{NormRand}$ with normal distribution in strategies *DEBest1JIte*r and *DERand1DIter*.

1. Input: $D, G_{\max}, NP \geq 4, F \in (0,1+), CR \in [0,1]$, and initial bounds: $\vec{x}^{(lo)}, \vec{x}^{(hi)}$.

2. Initialize: $\begin{cases} \forall i \leq NP \wedge \forall j \leq D : x_{i,j,G=0} = x_j^{(lo)} + rand_j[0,1] \bullet \left(x_j^{(hi)} - x_j^{(lo)}\right) \\ i = \{1,2,...,NP\},\ j = \{1,2,...,D\},\ G = 0,\ rand_j[0,1] \in [0,1] \end{cases}$

$\begin{cases} \text{3. While } G < G_{\max} \\ \quad \forall i \leq NP \begin{cases} \text{4. Mutate and recombine:} \\ \quad 4.1 \quad r_1, r_2, r_3 \in \{1,2,....,NP\}, \text{ randomly selected, except: } r_1 \neq r_2 \neq r_3 \neq i \\ \quad 4.2 \quad j_{rand} \in \{1,2,...,D\}, \text{ randomly selected once each } i \\ \quad 4.3 \quad \forall j \leq D, u_{j,i,G+1} = \begin{cases} x_{j,r_3,G} + F \cdot (x_{j,r_1,G} - x_{j,r_2,G}) \\ \qquad \text{if } (rand_j[0,1] < CR \vee j = j_{rand}) \\ x_{j,i,G} \quad \text{otherwise} \end{cases} \\ \text{5. Select} \\ \vec{x}_{i,G+1} = \begin{cases} \vec{u}_{i,G+1} & \text{if } f(\vec{u}_{i,G+1}) \leq f(\vec{x}_{i,G}) \\ \vec{x}_{i,G} & \text{otherwise} \end{cases} \end{cases} \\ G = G + 1 \end{cases}$

**Figure 60:** Canonical DE Schematic.

1. Input: $D, G_{\max}, NP \geq 4, F \in (0,1+), CR \in [0,1]$, and initial bounds: $\vec{x}^{(lo)}, \vec{x}^{(hi)}$.

2. Initialize: $\begin{cases} \forall i \leq NP \wedge \forall j \leq D : x_{i,j,G=0} = x_j^{(lo)} + rand_j[0,1] \bullet \left(x_j^{(hi)} - x_j^{(lo)}\right) \\ i = \{1,2,...,NP\},\ j = \{1,2,...,D\},\ G = 0,\ rand_j[0,1] \in [0,1] \end{cases}$

3. While $G < G_{\max}$

$\forall i \leq NP$

4. Mutate and recombine:

4.1 $r_1, r_2, r_3 \in \{1,2,....,NP\}$, randomly selected, except: $r_1 \neq r_2 \neq r_3 \neq i$

4.2 $j_{rand} \in \{1,2,...,D\}$, randomly selected once each $i$

4.3 $\forall j \leq D, u_{j,i,G+1} = \begin{cases} x_{j,r_3,G} + F \cdot (x_{j,r_1,G} - x_{j,r_2,G}) \\ \qquad \text{if } (rand_j[0,1] < CR \vee j = j_{rand}) \\ x_{j,i,G} \quad \text{otherwise} \end{cases}$

5. Select

$\vec{x}_{i,G+1} = \begin{cases} \vec{u}_{i,G+1} & \text{if } f(\vec{u}_{i,G+1}) \leq f(\vec{x}_{i,G}) \\ \vec{x}_{i,G} & \text{otherwise} \end{cases}$

$G = G + 1$

**Table 4:** Description of selected DE Strategies

# 4 Neural Computing

Neural Computing, e.g. Artificial Neural Networks, is one of the most interesting and rapidly growing areas of research, attracting researchers from a wide variety of scientific disciplines. Starting from the basics, Neural Computing covers all the major approaches, putting each in perspective in terms of their capabilities, advantages, and disadvantages.

An Artificial Neural Network (ANN) is an information processing paradigm that is inspired by the way of biological nervous systems, such as the brain, process information. The key element of this paradigm is the structure of the information processing system. It is composed of a large number of highly interconnected processing elements (neurones) working in unison to solve specific problems. ANNs, like people, learn by example. An ANN is configured for a specific application, such as pattern recognition or data classification, through a learning process. Learning in biological systems involves adjustments to the synaptic connections that exist between the neurones. This is true of ANNs as well.

## 4.1 The brain as an information processing system

The human brain contains about 10 billion nerve cells, or neurons. On average, each neuron is connected to other neurons through about 10 000 synapses. (The actual figures vary greatly, depending on the local neuroanatomy.) The brain's network of neurons forms a massively parallel information processing system. This contrasts with conventional computers in which a single processor executes a single series of instructions.

| | processing elements | element size | energy use | processing speed | style of computation | fault tolerant | learns | intelligent, conscious |
|---|---|---|---|---|---|---|---|---|
|  | $10^{14}$ synapses | $10^{-6}$ m | 30 W | 100 Hz | parallel, distributed | yes | yes | usually |
|  | $10^{8}$ transistors | $10^{-6}$ m | 30 W (CPU) | $10^{9}$ Hz | serial, centralized | no | a little | not (yet) |

**Table 5:** Description of selected DE Strategies (adapted from http://www.idsia.ch)

Against this, consider the time taken for each elementary operation: neurons typically operate at a maximum rate of about 100 Hz, while a conventional CPU carries out several hundred million machine level operations per second. Despite of being built with very slow hardware, the brain has quite remarkable capabilities:

- Its performance tends to degrade gracefully under partial damage. In contrast, most programs and engineered systems are brittle: if you remove some arbitrary parts, very likely the whole will cease to function.
- It can learn (reorganize itself) from experience.
- This means that partial recovery from damage is possible if healthy units can learn to take over the functions previously carried out by the damaged areas.
- It performs massively parallel computations extremely efficiently. For example, complex visual perception occurs within less than 100 ms, that is, 10 processing steps!

As a discipline of Artificial Intelligence, Neural Networks attempt to bring computers a little closer to the brain's capabilities by imitating certain aspects of information processing in the brain, in a highly simplified way. The comparison of computer and brain abilities is shown in Table 5.

The brain is not homogeneous. At the largest anatomical scale, we distinguish cortex, midbrain, brainstem, and cerebellum. Each of these can be hierarchically subdivided into many regions, and areas within each region, either according to the anatomical structure of the neural networks within it, or according to the function performed by them. The overall pattern of projections (bundles of neural connections) between areas is extremely complex, and only partially known. The best mapped (and largest) system in the human brain is the visual system, where the first 10 or 11 processing stages have been identified. We distinguish feedforward projections that go from earlier processing stages (near the sensory input) to later ones (near the motor output), from feedback connections that go in the opposite direction. In addition to these long-range connections, neurons also link up with many thousands of their neighbours. In this way they form very dense, complex local networks.

The basic computational unit in the nervous system is the nerve cell, or neuron. A biological neuron has, see Figure 61:

- Dendrites (inputs) a neuron
- Cell body
- Axon (output)

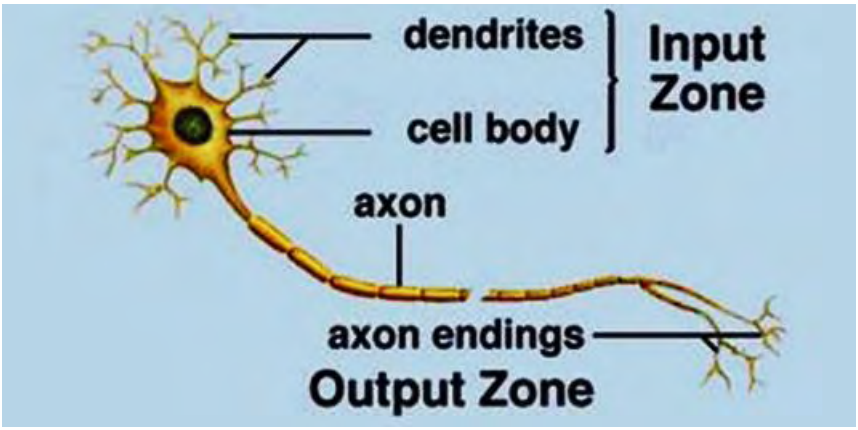



**Figure 61**: A biological neuron (adapted from http://www.idsia.ch

A neuron receives input from other neurons (typically many thousands). Inputs sum (approximately). Once input exceeds a critical level, the neuron discharges a spike – an electrical pulse that travels from the body, down the axon, to the next neuron(s) (or other receptors). This spiking event is also called depolarization, and is followed by a refractory period, during which the neuron is unable to fire.

The axon endings (Output Zone) almost touch the dendrites or cell body of the next neuron. Transmission of an electrical signal from one neuron to the next is effected by neurotransmittors, chemicals which are released from the first neuron and which bind to receptors in the second. This link is called a synapse. The extent to which the signal from one neuron is passed on to the next depends on many factors, e.g. the amount of neurotransmittors available, the number and arrangement of receptors, amount of neurotransmittors reabsorbed, etc.

Brains learn. From what we know of neuronal structures, one way brains learn is by altering the strengths of connections between neurons, and by adding or deleting connections between neurons. Furthermore, they learn "on-line", based on experience, and typically without the benefit of a benevolent teacher. The efficacy of a synapse can change as a result of experience, providing both memory and learning through long-term potentiation. One way this happens is through release of more neurotransmitters. Many other changes may also be involved, see Figure 62.

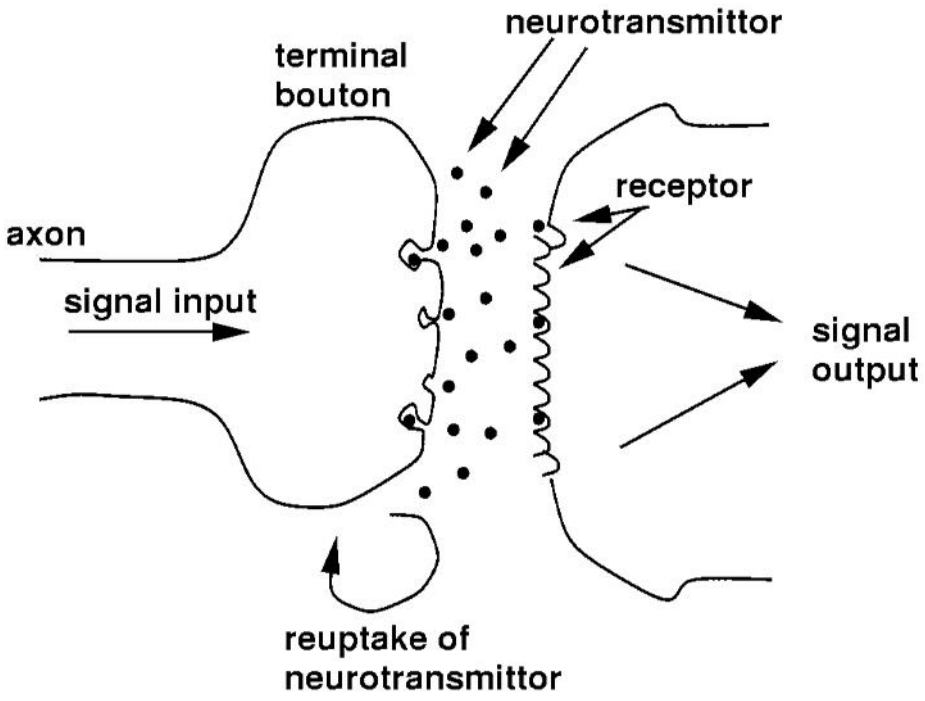



**Figure 62**: A biological neuron (adapted from http://www.idsia.ch)

## 4.2 Introduction to neural networks

An artificial neural network is a connectionist massively parallel system, inspired by the human neural system. Its units, neurons (Figure 63), are interconnected by connections called synapse. Each neuron, as the main computational unit, performs only a very simple operation: it sums its weighted inputs and applies a certain activation function on the sum. Such a value then represents the output of the neuron. However great such a simplification is (according to the biological neuron), it has been found as plausible enough and is successfully used in many types of ANN, (Fausett 1994).

A neuron $X_i$ obtains input signals $\boldsymbol{x}_i$ and relevant weights of connections $\boldsymbol{w}_i$, optionally a value called bias $b_i$ is added in order to shift the sum relative to the origin. The weighted sum of inputs is computed and the bias is added so that we obtain a value called stimulus or inner potential of the neuron $s_i$. After that it is transformed by an activation function $f$ into output value $o_i$ that is computed as it is shown in equations (see Figure 63):

$$s_i = \sum_{j=1}^{n} w_j x_j + b_i$$
$$o_i = \left(1 + e^{-s_i}\right)^{-1} ,$$

and may be propagated to other neurons as their input or be considered as an output of the network. Here, the activation function is a sigmoid, (Kondratenko and Kuperin 2003). The purpose of the activation function is to perform a threshold operation on the potential of the neuron.

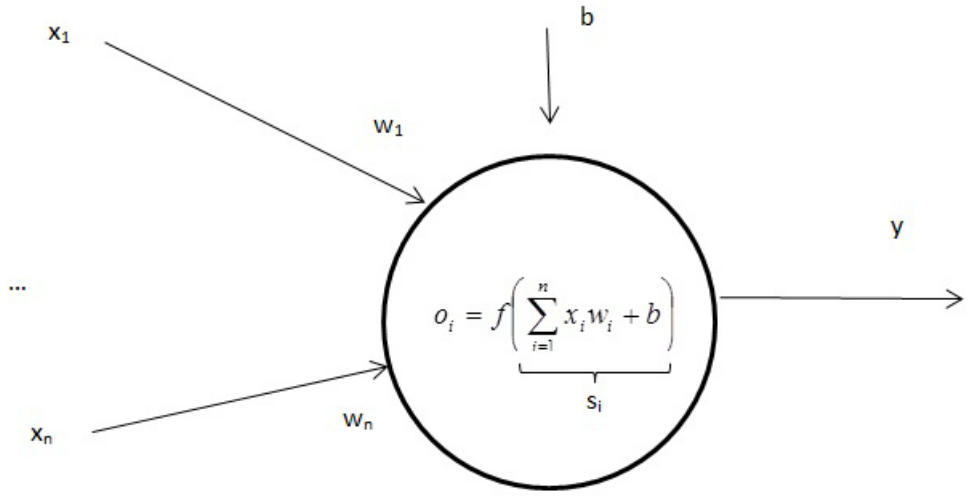



**Figure 63:** A simple artificial neuron

**Activation functions**

Most units in neural network transform their net inputs by using a scalar-to-scalar function called an *activation function*, yielding a value called the unit's activation. Except possibly for output units, the activation value is fed to one or more other units. Activation functions with a bounded range are often called squashing functions. Some of the most commonly used activation functions are the following (Fausett 1994).

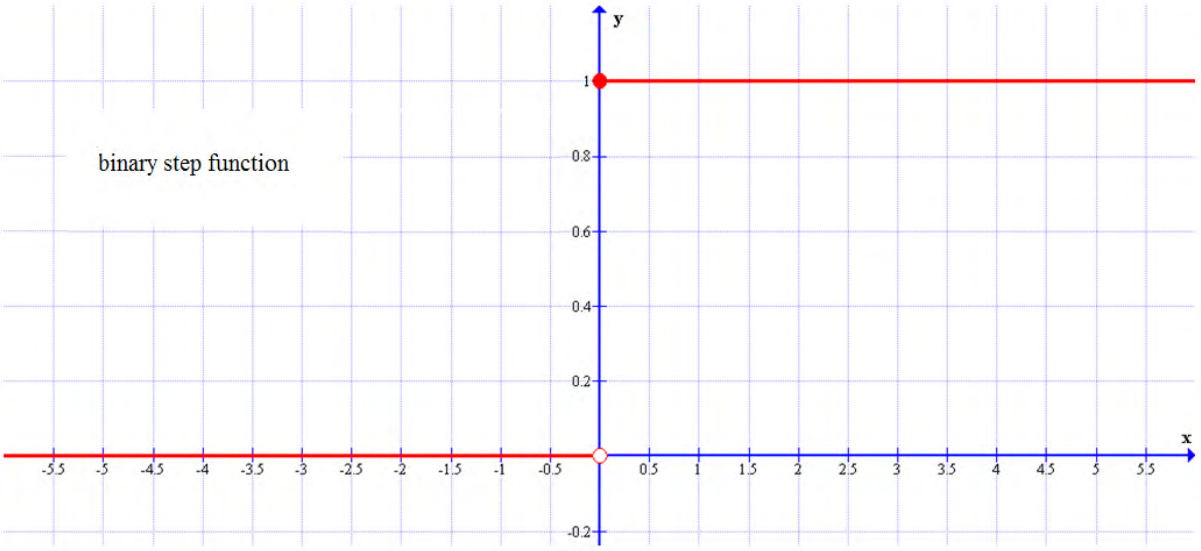
$$f(x)=\begin{cases}1 & x\geq 0\\ 0 & x<0\end{cases} \quad \text{binary step function}$$

$$f(x)=\begin{cases}1 & x>1\\ x & 0\leq x\leq 1\\ 0 & x<0\end{cases} \quad \text{saturated linear function}$$

$$f(x)=\frac{1}{1+e^{-x}} \quad \text{standard sigmoid}$$

$$f(x)=\frac{1-e^{-2x}}{1+e^{-2x}} \quad \text{hyperbolic tangent}$$

Graphs of these activation functions are shown in Figure 64.



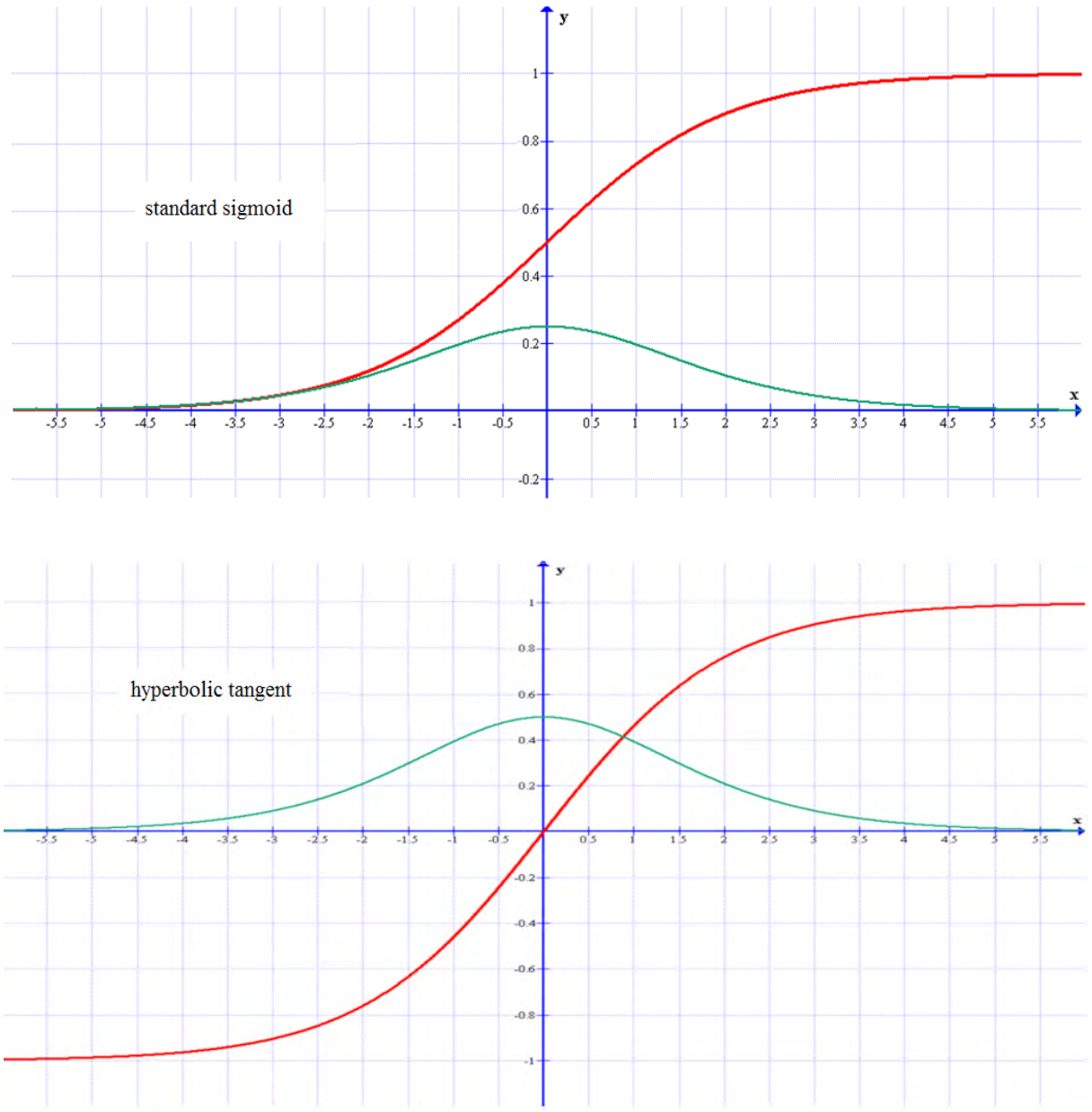




**Figure 64:** Graphs of activation functions.

### Network topologies

Network topologies focus on the pattern of connections between the units and the propagation of data. The basic models are the following:

- *Feed-forward networks* (Figure 65), where the data flow from input to output units is strictly feed-forward. The data processing can extend over multiple (layers of) units, but no feedback connections are present, that is, connections extending from outputs of units to inputs of units in the same layer or previous layers.

- *Recurrent networks* (Figure 66) contain feedback connections. Contrary to feed-forward networks, the dynamical properties of the network are important. In some cases, the activation values of the units undergo a relaxation process such that the network will evolve to a stable state in which these activations do not change anymore. In other applications, the change of the activation values of the output neurons is significant such that the dynamical behavior constitutes the output of the network.

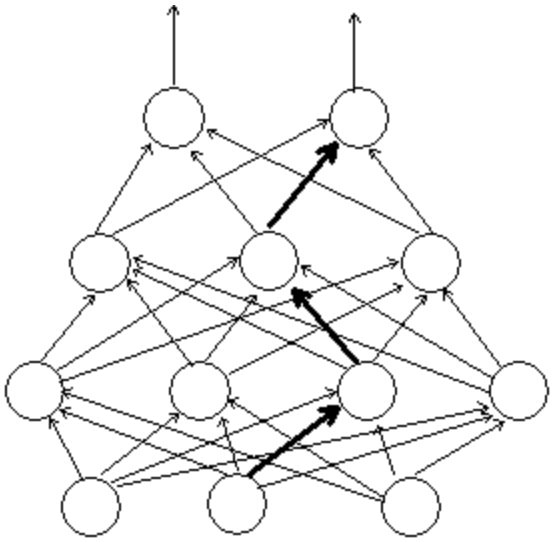


**Figure 65:** Feed-forward networks

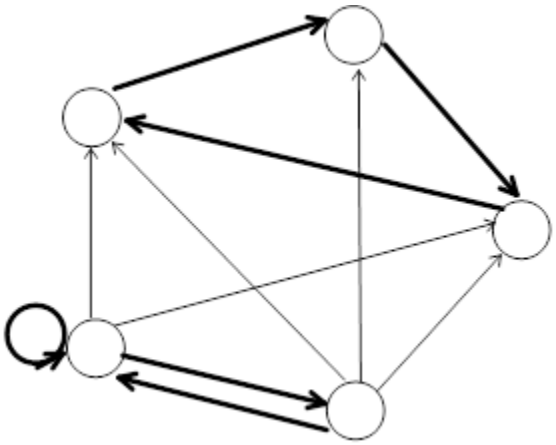


**Figure 66:** Recurrent networks

Classical examples of feed-forward networks are the Perceptron and Adaline. Examples of recurrent networks are Hopfield nets.

**Training of artificial neural networks**

A neural network has to be configured such that the application of a set of inputs produces (either 'direct' or via a relaxation process) the desired set of outputs. Various methods to set the strengths of the connections exist. One way is to set the weights explicitly, using a priori knowledge. Another way is to 'train' the neural network by feeding it teaching patterns and letting it change its weights according to some learning rule.

We can categorize the learning situations in two distinct sorts. These are:

- *Supervised learning* or *associative learning* in which the network is trained by providing it with input and matching output patterns. These input-output pairs can be provided by an external teacher, or by the system which contains the network (self-supervised).
- *Unsupervised learning* or *self-organization* in which an (output) unit is trained to respond to clusters of pattern within the input. In this paradigm the system is supposed to discover statistically salient features of the input population. Unlike the supervised learning paradigm, there is no a priori set of categories into which the patterns are to be classified; rather the system must develop its own representation of the input stimuli.

**Hebb rule**

Both learning paradigms discussed above result in an adjustment of the weights of the connections between units, according to some modification rule. Virtually all learning rules for models of this type can be considered as a variant of the Hebbian learning rule suggested by Hebb in the classic book *Organization of Behaviour* (Hebb 1949). The Hebb rule determines the change in the weight connection from $u_i$ to $u_j$ by $\Delta w_{ij} = \alpha * y_i * y_j$, where $\alpha$ is the learning rate and $y_i$, $y_j$ represent the activations of $u_i$ and $u_j$ respectively. Thus, if both $u_i$ and $u_j$ are activated the weight of the connection from $u_i$ to $u_j$ should be increased.

Examples can be given of input/output associations which can be learned by a two-layer Hebb rule pattern associator. In fact, it can be proved that if the set of input patterns used in training are mutually orthogonal, the association can be learned by a two-layer pattern associator using Hebbian learning. However, if the set of input patterns are not mutually orthogonal, interference may occur and the network may not be able to learn associations. This limitation of Hebbian learning can be overcome by using the delta rule.

**Delta rule**

The delta rule (Russell 2005), also called the Least Mean Square (LMS) method, is one of the most commonly used learning rules. For a given input vector, the output vector is compared to the correct answer. If the difference is zero, no learning takes place; otherwise, the weights are adjusted to reduce this difference. The change in weight from $u_i$ to $u_j$ is given by: $\Delta w_{ij} = \alpha * y_i * e_j$, where $\alpha$ is the learning rate, $y_i$ represents the activation of $u_i$ and $e_j$ is the difference between the expected output and the actual output of $u_j$. If the set of input patterns form a linearly independent set then arbitrary associations can be learned using the delta rule.

This learning rule not only moves the weight vector nearer to the ideal weight vector, it does so in the most efficient way. The delta rule implements a gradient descent by moving the weight vector from the point on the surface of the paraboloid down toward the lowest point, the vertex.

In the case of linear activation functions where the network has no hidden units, the delta rule will always find the best set of weight vectors. On the other hand, that is not the case for hidden units. The error surface is not a paraboloid and so does not have a unique minimum point. There is no such powerful rule as the delta rule for networks with hidden units. There have been a number of theories in response to this problem. These include the generalized delta rule and the unsupervised competitive learning model.

Generalizing the ideas of the delta rule, consider a hierarchical network with an input layer, an output layer and a number of hidden layers. We consider only the case where there is one hidden layer. The network is presented with input signals which produce output signals that act as input to the middle layer. Output signals from the middle layer in turn act as input to the output layer to produce the final output vector. This vector is compared to the desired output vector. Since both the output and the desired output vectors are known, we can calculate differences between both outputs and get an error of neural network. The error is backpropagated from the output layer through the middle layer to the unit which are responsible for generating that output. The delta rule can be used to adjust all the weights. More details are presented in (Fausett 1994).

## 4.3 The perceptron

The perceptron (Figure 67) is a simplest type of artificial neural networks that is linear and based on a threshold $\theta$ transfer function. The perceptron can only classify linearly separable cases with a binary target 1 or 0.

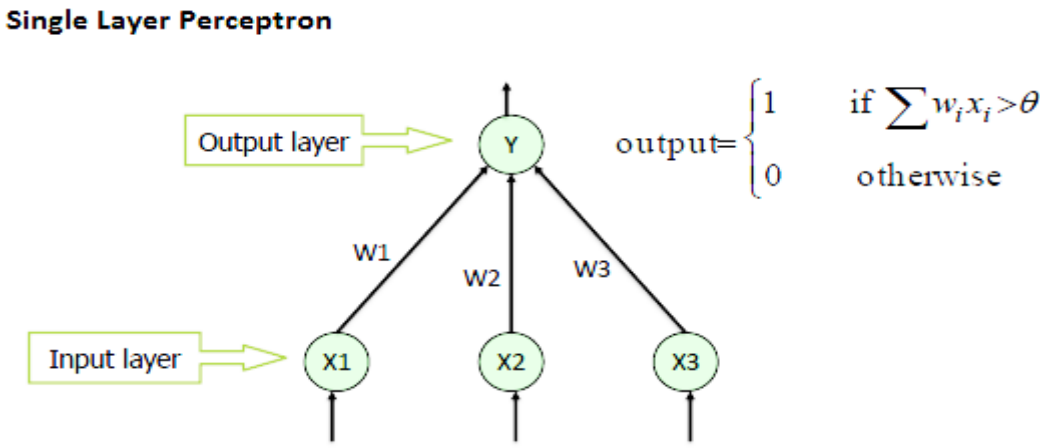



**Figure 67:** The perceptron (adapted from http://www.saedsayad.com/artificial_neural_network_bkp.htm)

The perceptron algorithm was invented in 1957 at the Cornell Aeronautical Laboratory by Frank Rosenblatt (Beale and Jackson 1992).

The single layer perceptron does not have a priori knowledge, so the initial weights are assigned randomly. The perceptron sums all the weighted inputs and if the sum is above the threshold (some predetermined value), it is said to be activated (output=1).

Output

$$w_1 x_1 + w_2 x_2 + \ldots + w_n x_n > \theta \Rightarrow 1$$

$$w_1 x_1 + w_2 x_2 + \ldots + w_n x_n \leq \theta \Rightarrow 0$$

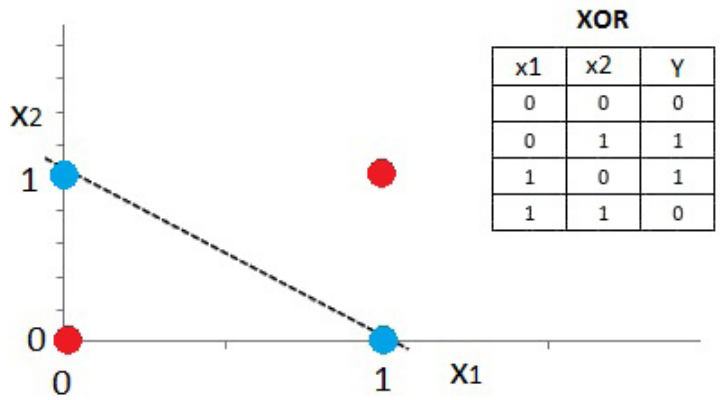
The input values are presented to the perceptron, and if the predicted output is the same as the desired output, then the performance is considered satisfactory and no changes to the weights are made. However, if the output does not match the desired output, then the weights need to be changed to reduce the error.

Perceptron Weight Adjustment is the following:

$$\Delta w = \alpha \times d \times x$$

$\alpha$ is a learning rate, usually less than 1,
$d$ is a "predicted output – desired output",
$x$ represents input data.

As the perceptron is a linear classifier and if the cases are not linearly separable, the learning process will never reach a point where all the cases are classified properly. The most famous example of the inability of perceptron to solve problems with linearly non-separable cases is the XOR problem (Figure 68). However, a multi-layer perceptron using the backpropagation algorithm can successfully classify the XOR data.




XOR

| x1 | x2 | Y |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

**Figure 68:** XOR problem (adapted from http://www.saedsayad.com/artificial_neural_network_bkp.htm)

*Example*

A perceptron for the AND function: binary inputs, binary targets. For simplicity, we take learning rate $\alpha = 1$ and threshold $\theta = 0{,}2$.

| time | INPUT | | OUTPUT | | | WEIGHT CHANGES | | | WEIGHTS | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | $x_1$ | $x_2$ | *NET* | *OUT* | *TARGET* | $\Delta w_1$ | $\Delta w_2$ | $\Delta b$ | $w_1$ | $w_2$ | $b$ |
| 0 | | | | | | | | | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 1 | 0 | 2 | 1 | -1 | -1 | 0 | -1 | 0 | 1 | 0 |
| 3 | 0 | 1 | 1 | 1 | -1 | 0 | -1 | -1 | 0 | 0 | -1 |
| 4 | 0 | 0 | -1 | -1 | -1 | 1 | 0 | 0 | 0 | 0 | -1 |
| ... | | | | | | | | | | | |
| 37 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 2 | 3 | -4 |
| 38 | 1 | 0 | -2 | -1 | -1 | 0 | 0 | 0 | 2 | 3 | -4 |
| 39 | 0 | 1 | -1 | -1 | -1 | 0 | 0 | 0 | 2 | 3 | -4 |
| 40 | 0 | 0 | -4 | -1 | -1 | 0 | 0 | 0 | 2 | 3 | -4 |

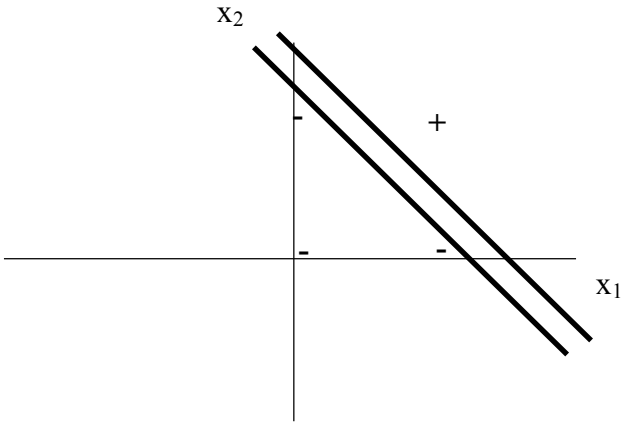



**Figure 69:** Final decision boundaries for AND function in perceptron learning

The positive response is given by all points such that: $2x_1 + 3x_2 - 4 > 0.2$

with boundary line: $x_2 = -\frac{2}{3}x_1 + \frac{7}{5}$.

The negative response is given by all points such that: $2x_1 + 3x_2 - 4 < -0.2$

with boundary line: $x_2 = -\frac{2}{3}x_1 + \frac{19}{15}$.

## 4.4 Multilayer networks

Many authors agree that multilayer feedforward neural networks (Figure 70) belong to the most common ones in practical use. Usually a fully connected variant is used, so that each neuron from the *n-th* layer is connected to all neurons in the *(n+1)-th* layer, but it is not necessary and in general some connections may be missing. There are also no connections between neurons of the same layer. A subset of input units has no input connections from other units; their states are fixed by the problem. Another subset of units is designated as output units; their states are considered the result of the computation. Units that are neither input nor output are known as hidden units, (Hertz and Kogh 1991).

Each problem specifies a training set of associated pairs of vectors for the input units and output units. The full specification of a network to solve a given problem involves enumerating all units, the connections between them, and setting the weights on those connections. The first two tasks are commonly solved in an ad hoc or heuristic manner, while the final task is usually accomplished with the aid of a learning algorithm, such as backpropagation. This algorithm belongs to a group called "gradient descent methods". An intuitive definition is that such an algorithm searches for the global minimum of the weight landscape by descending downhill in the most precipitous direction (Figure 71).

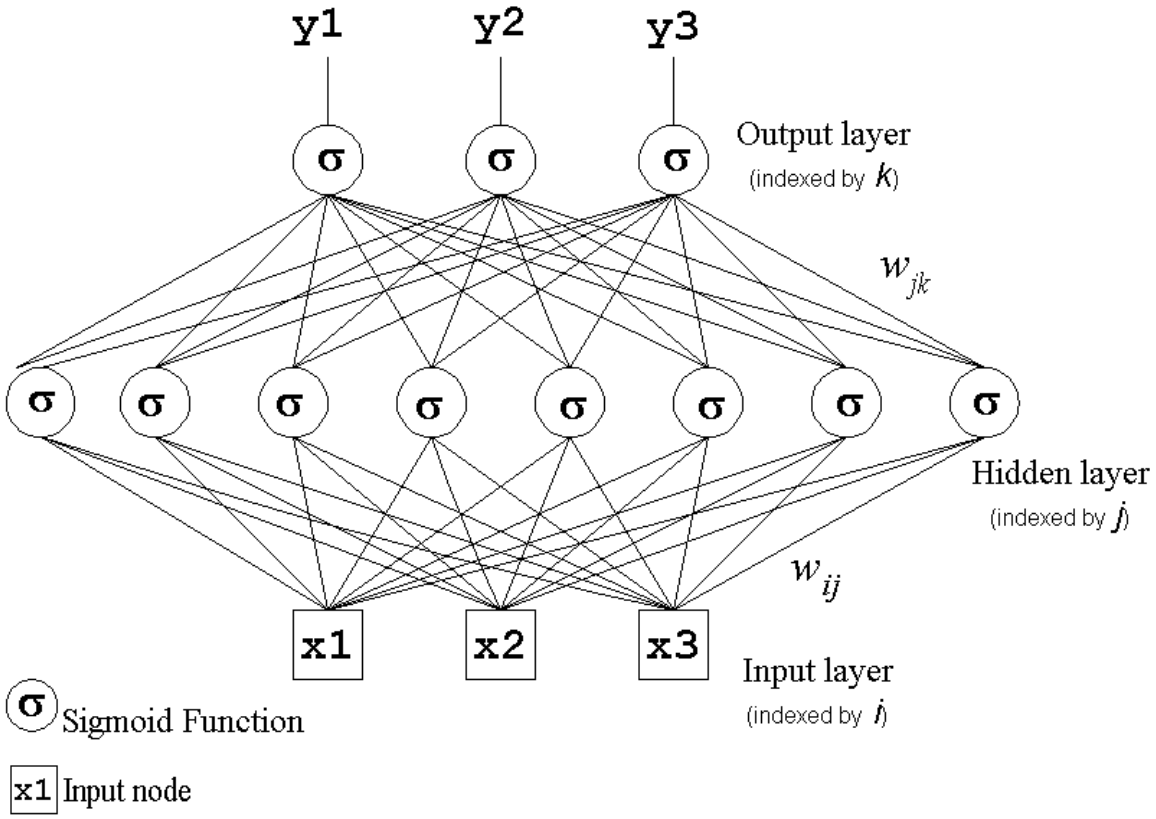



**Figure 70:** A multilayer feedforward neural network (adapted from http://homepages.gold.ac.uk/nikolaev/311multi.htm)

The initial position is set at random (note that there is no a priori knowledge about the shape of the landscape) selecting the weights of the network from some range (typically from 1 to 1 or from 0 to 1). It is obvious that the initial position on the weight landscape greatly influences both the length and the path made when seeking the global minimum. In some cases it is even impossible to get to the optimal position due to the occurrence of some deep local minima. Considering the different points, it is clear, that backpropagation using a fully connected neural network is not a deterministic algorithm. Now, a more formal definition of the backpropagation algorithm (for a three layer network) is presented, (Fausett 1994).

1. The input vector is presented to the network.
2. The feedforward is performed, so that each neuron computes its output following the formula over neurons in previous layer:

$$o_i = \frac{1}{1 + e^{\left(-\sum_{j=1}^{n} x_j w_j + b\right)}}$$

3. The error on the output layer is computed for each neuron using the desired output ($y_j$) on the same neuron:

$$err_j^0 = o_j(1-o_j)(y_j - o_j)$$

4. The error is propagated back to the hidden layer over all the hidden neurons ($h_i$) and weights between each of them and over all neurons in the output layer:

$$err_i^h = h_i(1-h_i)\sum_{j=1}^{r} err_j^0 w_j^0$$

5. Having values $err_j^0$ and $err_i^h$ computed, the weights from the hidden to the output layer and from the input to the hidden layer can be adjusted using the following formulas

$$w_{ij}^0(t+1) = w_j^0(t) + \alpha\ err_j^0 h_i$$
$$w_j^h(t+1) = w_j^h(t) + \alpha\ err_j^h x_i$$

where α is the learning coefficient and $x_i$ is the *i-th* neuron in the input layer.

6. All the preceding steps are repeated until the total error of the network over all training pairs does not fall under certain level, where $m$ is number of output neurons.

$$E = \frac{1}{2}\sum_{i=1}^{m}(y_i - o_i)^2$$

The formulas in step three and four are products of derivation of the error function on each node. A detailed explanation of this derivation as well as of the complete algorithm can be found in (Hertz and Kogh 1991).

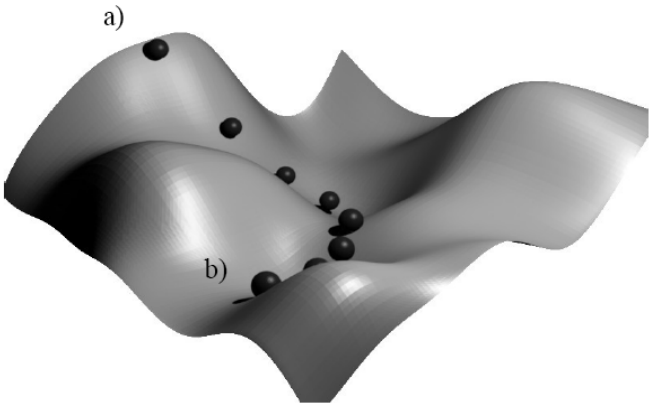



**Figure 71:** An intuitive approach to the gradient descent method, looking for the global minimum: a) is the starting point, b) is the final one.

## 4.5 Kohonen self-organizing maps

Kohonen Self-Organizing Maps (or just Self-Organizing Maps, or SOMs for short), are a type of neural network. They were developed in 1982 by Tuevo Kohonen, a professor emeritus of the Academy of Finland. Self-Organizing Maps are aptly named "Self-Organizing" because no supervision is required. SOMs learn on their own through unsupervised competitive learning. "Maps" is because they attempt to map their weights to conform to the given input data. The nodes in a SOM network attempt to become like the inputs presented to them. In this sense, this is how they learn.

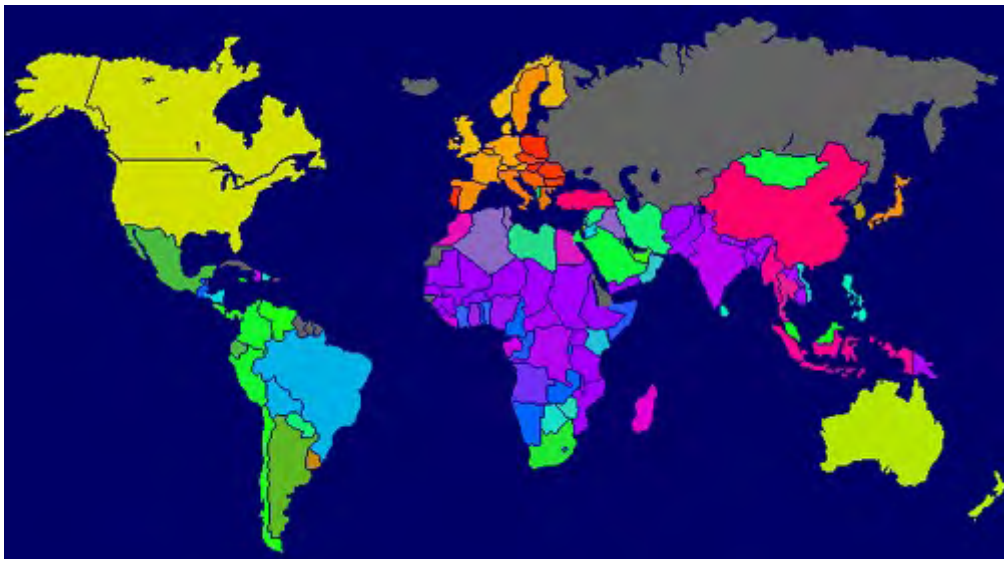


**Figure 72:** A map of the world quality-of-life (adapted from http://www.shy.am).

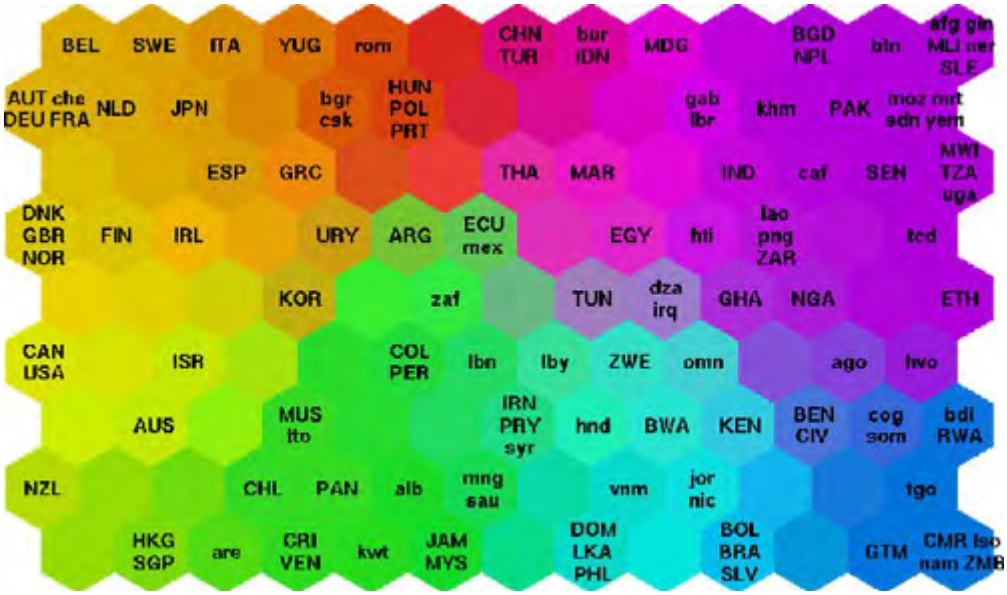



**Figure 73:** SOM of world quality-of-life (adapted from http://www.shy.am).

SOM can also be called "Feature Maps", as in Self-Organizing Feature Maps. Retaining principle 'features' of the input data is a fundamental principle of SOMs, and one of the things that makes them so valuable. Specifically, the topological relationships between input data are preserved when mapped to a SOM network. This has a pragmatic value of representing complex data. Figure 72 represents a map of the world quality-of-life. Yellows and oranges represent wealthy nations, while purples and blues are the poorer nations. From this view, it can be difficult to visualize the relationships between countries. However, represented by a SOM as shown if Figure 73, it is much easier to see what is going on. Here we can see the United States, Canada, and Western European countries, on the left side of the network, being the wealthiest countries. The poorest countries, then, can be found on the opposite side of the map (at the point farthest away from the richest countries), represented by the purples and blues. Figure 73 is a hexagonal grid. Each hexagon represents a node in the neural network. This is typically called a unified distance matrix, and is probably the most popular method of displaying SOMs. Another intrinsic property of SOMs is known as vector quantization. This is a data compression technique. SOMs provide a way of representing multidimensional data in a much lower dimensional space – typically one or two dimensions. This aides in their visualization benefit, as humans are more proficient at comprehending data in lower dimensions than higher dimensions, as can be seen in the comparison of Figure 72 to Figure 73. The above examples show how SOMs are a valuable tool in dealing with complex or vast amounts of data. In particular, they are extremely useful for the visualization and representation of these complex or large quantities of data in manner that is most easily understood by the human brain.

### Structure of a SOM

The structure of a SOM is fairly simple, and is best understood with the use of an illustration such as Figure 74.

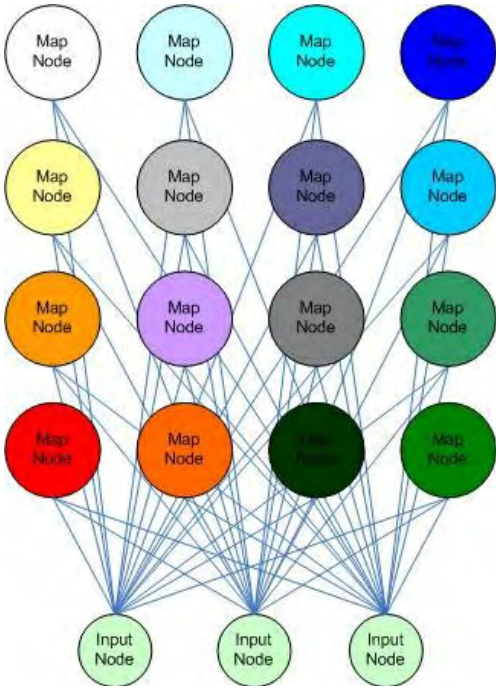



**Figure 74:** Structure of a SOM (adapted from http://www.shy.am).

Figure 74 is a 4×4 SOM network (4 nodes down, 4 nodes across). It is easy to overlook this structure as being trivial, but there are a few key things to notice. First, each map node is connected to each input node. For this small 4×4 node network, that is 4×4×3=48 connections. Secondly, notice that map nodes are not connected to each other. The nodes are organized in this manner, as a 2-D grid makes it easy to visualize the results. This representation is also useful when the SOM algorithm is used. In this configuration, each map node has a unique (*i, j*) coordinate. This makes it easy to reference a node in the network, and to calculate the distances between nodes. Because of the connections only to the input nodes, the map nodes are oblivious as to what values their neighbours have. A map node will only update its' weights (explained next) based on what the input vector tells it.

The following relationships describe what a node essentially is:

1. $network \subset mapNode \subset float\ weights\ [numWeights]$
2. $inputVectors \subset inputVector \subset float\ weights\ [numWeights]$

The *first* relationship says that the network (the 4×4 grid above) contains map nodes. A single map node contains an array of floats, or its weights. *numWeights* will become more apparent during application discussion. The only other common item that a map node should contain is its (*i, j*) position in the network.

The *second* relationship says that the collection of input vectors (or input nodes) contains individual input vectors. Each input vector contains an array of floats, or its' weights. Note that *numWeights* is the same for both weight vectors. The weight vectors must be the same for map nodes and input vectors or the algorithm will not work.

**The SOM algorithm**

The Self-Organizing Map algorithm can be broken up into 6 steps (Beale and Jackson 1992).

1. Each node's weights are initialized.
2. A vector is chosen at random from the set of training data and presented to the network.
3. Every node in the network is examined to calculate which ones' weights are most like the input vector. The winning node is commonly known as the *Best Matching Unit* (BMU).

$$DistFromInput^2 = \sum_{i=0}^{n} \left(I_i - W_i\right)^2$$

*I* is current input vector
*W* is node's weight vector
*n* is number of weights

4. The radius of the neighborhood of the BMU is calculated. This value starts large. Typically it is set to be the radius of the network, diminishing each time-step.

*Radius of the neighborhood:*

$$\sigma(t) = \sigma_0 e^{(-t/\lambda)}$$

*t* is current iteration
$\lambda$ is time constant
$\sigma_0$ is radius of the map

*Time constant:*
$\lambda$ = *numIterations* / *mapRadius*

5. Any nodes found within the radius of the BMU calculated in 4) are adjusted to make them more like the input vector (Equation 3a, 3b).

   *New weight of a node:*

   $$W(t+1) = W(t) + \Theta(t)L(t)(I(t) - W(t))$$

   *Learning rate*:

   $$L(t) = L_0 e^{[-t/\lambda]}$$

   The closer a node is to the BMU, the more its weights are altered:
   Distance from BMU:

   $$\Theta(t) = e^{\left(-distFromBMU^2/(2\sigma^2(t))\right)}$$

6. Repeat 2) for *N* iterations.

There are some things to note about these formulas. Equation from step 3 represents simply the Euclidean distance formula, squared. It is squared because we are not concerned with the actual numerical distance from the input. We just need some sort of uniform scale in order to compare each node to the input vector. This equation provides that eliminating the need for a computationally expensive square root operation for every node in the network.

Equations from step 4 utilize exponential decay. At $t$=0 they are at their max. As $t$ (the current iteration number) increases, they approach zero. This is exactly what we want. The radius should start out as the radius of the lattice, and approach zero, at which time the radius is simply the BMU node (see Figure 75). The time constant value is almost arbitrary and can be chosen. This provides a good value, though, as it depends directly on the map size and the number of iterations to perform.

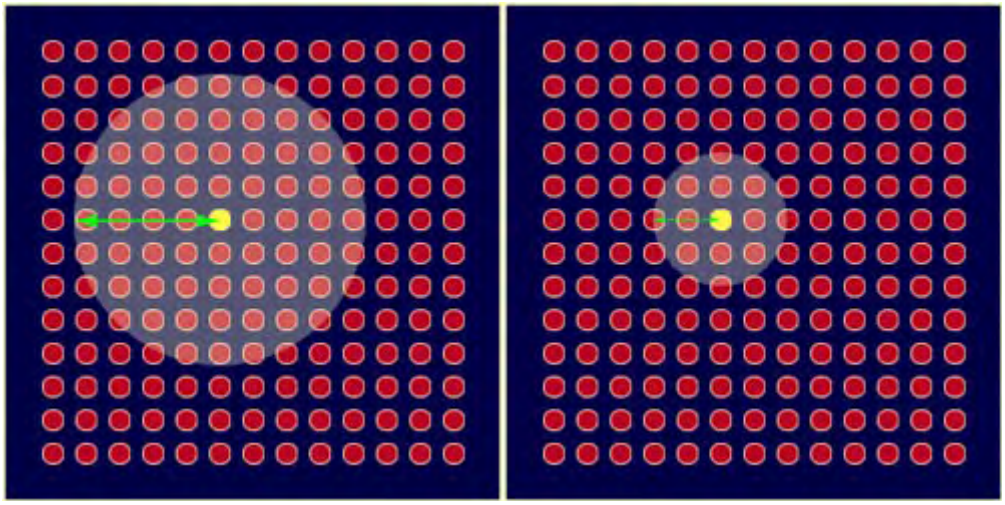


**Figure 75:** Radius of the neighbourhood (adapted from http://www.shy.a)

Equation from step 5 is the main learning function. $W(t+1)$ is the new, 'educated', weight value of the given node. Over time, this equation essentially makes a given node weight more like the currently selected input vector, I. A node that is very different from the current input vector will learn more than a node very similar to the current input vector. The difference between the node weight and the input vector are then scaled by the current learning rate of the SOM, and by $\Theta(t)$.

$\Theta(t)$ is used to make nodes closer to the BMU learn more than nodes on the outskirts of the current neighborhood radius. Nodes outside of the neighborhood radius are skipped completely. *distFromBMU* is the actual number of nodes between the current node and the BMU, easily calculated as: $distFromBMU^2 = (bmuI - nodeI)^2 + (bmuJ - nodeJ)^2$

This can be done since the node network is just a 2-D grid of nodes. With this in mind, nodes on the very fringe of the neighbourhood radius will learn some fraction less 1.0. As *distFromBMU* decreases, $\Theta(t)$ approaches 1.0. The BMU itself will have a distFromBMU equal to 0, which gives $\Theta(t)$ it's maximum value of 1.0. Again, this Euclidean distance remains squared to avoid the square root operation.

There exists a lot of variations regarding the equations used with the SOM algorithm. There is also a lot of research being done on the optimal parameters. Some things of particular heavy debate are the number of iterations, the learning rate, and the neighborhood radius. It has been suggested by Kohonen himself, however, that the training should be split into two phases. Phase 1 will reduce the learning coefficient from 0.9 to 0.1, and the neighbourhood radius from half the diameter of the lattice to the immediately surrounding nodes. Phase 2 will reduce the learning rate from 0.1 to 0.0, but over double or more the number of iterations in Phase 1. In Phase 2, the neighbourhood radius value should remain fixed at 1 (the BMU only). Analysing these parameters, Phase 1 allows the network to quickly 'fill out the space', while Phase 2 performs the 'fine-tuning' of the network to a more accurate representation.

*Example – Colour Classification*

Colour classification SOMs only use three weights per map and input nodes. These weights represent the (R,G,B) triplet for the colour. For example, colours may be presented to the network – (1,0,0) for red, (0,1,0) for green, etc. The goal for the network here is to learn how to represent all of these input colours on its 2-D grid while maintaining the intrinsic properties of a SOM such as retaining the topological relationships between input vectors. With this in mind, if dark blue and light blue are presented to the SOM, they should end up next to each other on the network grid.

To illustrate the process, we will step through the algorithm for the colour classification application. Step 1 is the initialisation of the network. Figure 76 shows a newly initialised network. Each square is a node in the network.

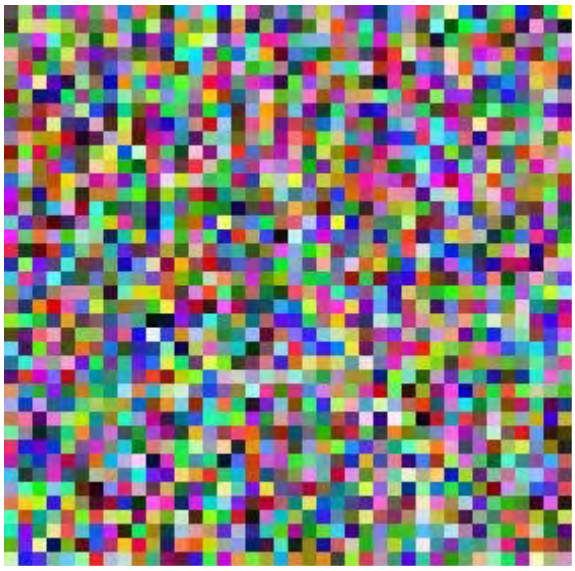


**Figure 76:** An initialised network (adapted from http://www.shy.am)

*Step* 1. The initialisation method used here is to assign a random value between 0.0 and 1.0 for each component (r, g, and b) of each node.

*Step 2* is to choose a vector at random from the input vectors. Eight input vectors are used in this example, ranging from red to yellow to dark green.

*Step 3* goes through every node and finds the BMU, as described earlier. Figure 77 shows the BMU being selected in the 4x4 network.

*Step 4* of the algorithm calculates the neighbourhood radius. This is also shown in Figure 77. All the nodes tinted red are within the radius. Step 5 then applies the learning functions to all of these nodes. It is based on their distance from the BMU. The BMU (dark red) learns the most, while nodes on the outskirts of the radius (light pink) learn the least. Nodes outside of the radius (white) don't learn at all. We then go back to Step 2 and repeat. Figure 78 shows a trained SOM, representing all eight input colours. Notice how light green is next to dark green, and red is next to orange. An ideal map would probably have light blue next to dark blue. This is where the *Error Map* comes into play, which is described next.

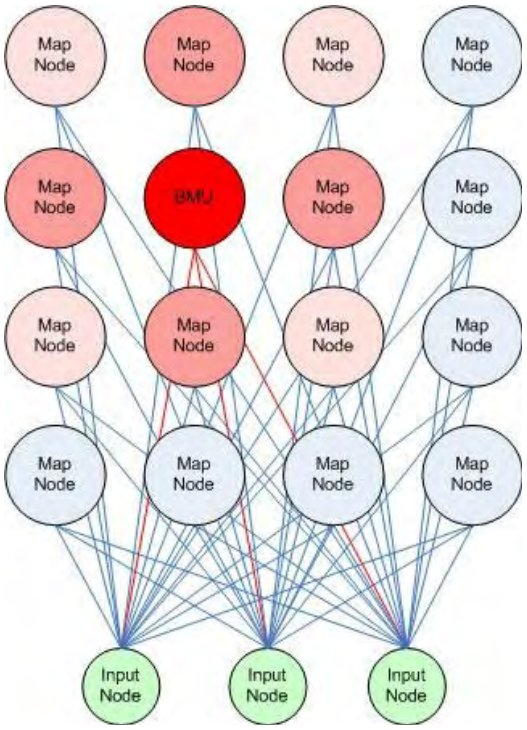



**Figure 77:** Best Matching Unit (BMU) (adapted from http://www.shy.am)

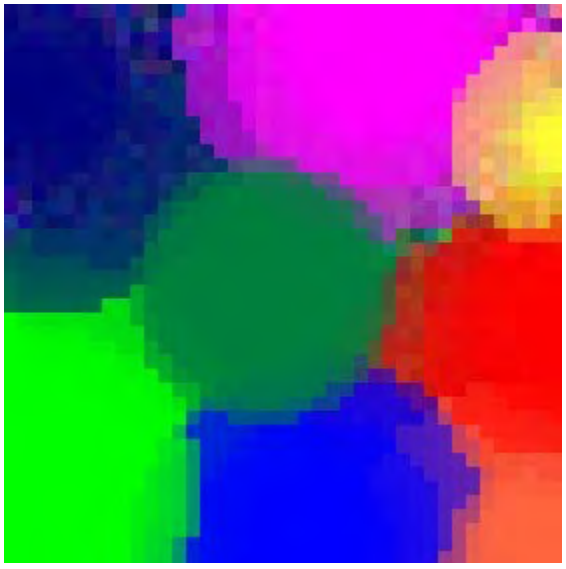


**Figure 78:** The BMU window (adapted from http://www.shy.am)

There are two other windows in the colour classification application. These are the BMU Window and the Error Map. These windows are not active until after the network is trained. First we describe the BMU window. Upon successful SOM training, this window will show small white dots. These white dots represent the N most frequently used BMU nodes, where N is the number of input vectors (unless N < # of iterations. Then, N = # of iterations). These nodes have been deemed to be a BMU the most times out of all the nodes in the network, presenting the least distance possible between a map node and the selected input vector for the given iteration. Figure 79 shows this window. Notice how Figure 79 could be placed on top of Figure 78, and the dots would correspond to the centers of the circles.

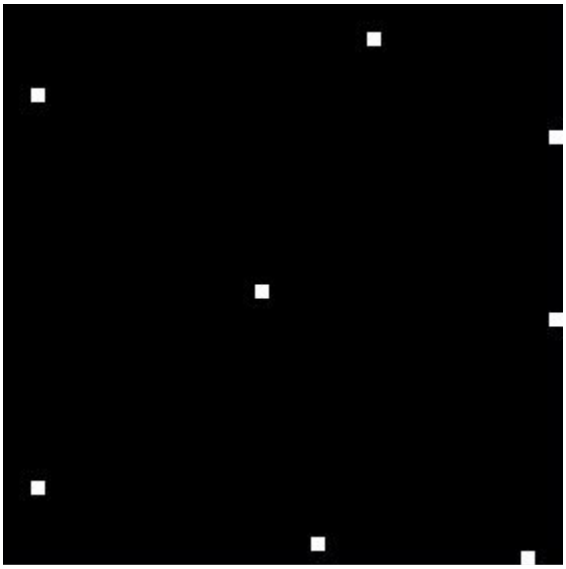


**Figure 79:** Dots correspond to the centres of the circles in Figure 76 (adapted from http://www.shy.am).

Next, the Error Map is calculated (Figure 79). Each time a SOM is trained, it can produce a completely different result given the same input data. This is because the network is initialised with random colours, presenting a unique setup prior to each training session. Also, this occurs because input vectors to be presented to the network are chosen at random. With this in mind, some SOMs may turn out 'better' than others, where 'better' is a measure of how well the topological data is preserved.

**Figure 80:** Error map corresponding with Figure 78 (adapted from http://www.shy.am).

To calculate an error map, loop through every map node of the network. Add up the distance (not the physical distance, but the weight distance. This is exactly the same as how the BMU is calculated) from the node we are currently evaluating, to each of its neighbours. Average this distance. Multiply this by 3 (the number of weights used), assuming no square root is used to calculate the distance between adjacent nodes. If the square root operation is used, multiply by $\sqrt{3}$ instead. Assign this value to the node. This gives each map node a nice value between 0.0 and 1.0. These values can then be used as the R = G = B values for each square of the Error Map window. Pure white represents the maximum possible distance between adjacent nodes, while black shows that adjacent nodes are all the same colour. Shades of gray in between give an even finer explanation, with darker grays being a better map than a map with light grays. Figure 80 shows an example. Notice the lines and how they line up with Figure 78.

## 4.6 Hopfield networks

A Hopfield network is a form of recurrent artificial neural network invented by John Hopfield. Hopfield nets serve as content-addressable memory systems with binary (bipolar) threshold nodes. They are guaranteed to converge to a local minimum, but convergence to a false pattern (wrong local minimum) rather than the stored pattern (expected local minimum) can occur. Hopfield networks also provide a model for understanding human memory.

Figure 81 shows a diagram representing a facial recognition system using a Hopfield network. The pixels are initially converted into black and white and the background is removed. Afterwards, the image is given to the network, which will eventually converge to the image on the right that the network used to train with.

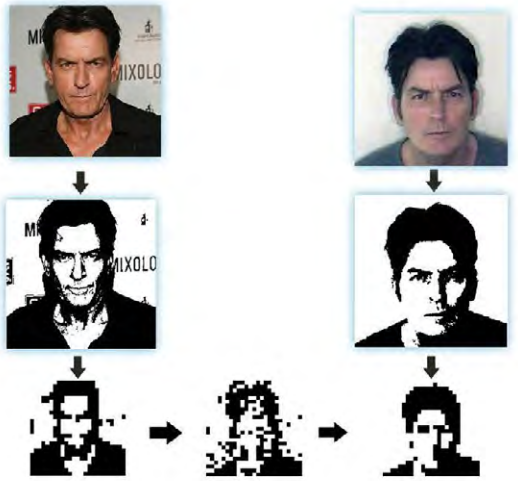


**Figure 81:** A diagram representing a facial recognition system using a Hopfield network (adapted from http://en.wikipedia.org/wiki/File:Face_recognition_with_hopfield_network.jpg

The units in Hopfield nets only take on two different values for their states and the value is determined by whether or not the units' input exceeds their threshold. Hopfield nets normally have units that take on values of 1 and -1 resp. 0 and 1.

Every two units $i$ and $j$ of a Hopfield network have a connection that is described by the connectivity weight $w_{ij}$. In this sense, the Hopfield network can be formally described as a complete undirected graph. The connections in a Hopfield net typically have the following restrictions:

- $w_{ii} = 0, \quad \forall i$ (no unit has a connection with itself)
- $w_{ij} = w_{ji}, \quad \forall i, j$ (connections are symmetric)

The requirement that weights be symmetric is typically used, as it will guarantee that the energy function decreases monotonically while following the activation rules, and the network may exhibit some periodic or chaotic behaviour if non-symmetric weights are used.

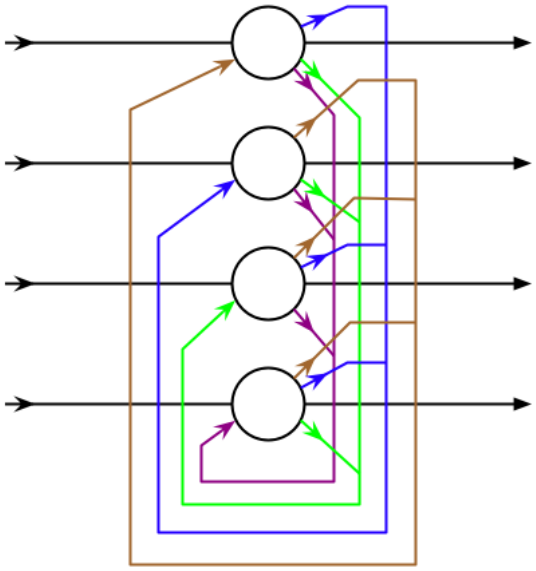


**Figure 82:** Hopfield network.
(adapted from http://upload.wikimedia.org/wikipedia/commons/9/95/Hopfield-net.png)

Updating one unit (node in the graph simulating the artificial neuron) in the Hopfield network is performed using the following rule:

$$s_i = \begin{cases} 1 & \text{if } \sum_j w_j s_j > \theta_i \\ -1 & \text{otherwise.} \end{cases}$$

where:

$w_{ij}$ is the strength of the connection weight from unit j to unit i (the weight of the connection).
$s_j$ is the state of unit $j$.
$\theta_i$ is the threshold of unit $i$.

Updates in the Hopfield network can be performed in two different ways:

- **Asynchronous**: Only one unit is updated at a time. This unit can be picked at random, or a pre-defined order can be imposed from the very beginning.
- **Synchronous**: All units are updated at the same time. This requires a central clock to the system in order to maintain synchronisation. This method is less realistic, since biological or physical systems lack a global clock that keeps track of time.

Neurons attract or repel each other. The weight between two units has a powerful impact upon the values of the neurons. Consider the connection weight $w_{ij}$ between two neurons i and j. If $w_{ij} > 0$, the updating rule implies that:

- when $s_j = 1$, the contribution of $j$ in the weighted sum is positive. Thus, $s_i$ is pulled by $j$ towards its value $s_j = 1$
- when $s_j = -1$, the contribution of $j$ in the weighted sum is negative. Then again, $s_i$ is pulled by $j$ towards its value $s_j = -1$

Thus, the values of neurons $i$ and $j$ will converge if the weight between them is positive. Similarly, they will diverge if the weight is negative.

**Energy**

Hopfield nets have a scalar value associated with each state of the network referred to as the "energy" (Figure 83), $E$, of the network, where:

$$E = -\frac{1}{2}\sum_{i,j} w_{ij} s_i s_j + \sum_i \theta_i s_i$$

This value is called the "energy" because the definition ensures that when units are randomly chosen to update, the energy $E$ will either lower in value or stay the same. Furthermore, under repeated updating the network will eventually converge to a state which is a local minimum in the energy function. Thus, if a state is a local minimum in the energy function, it is a stable state for the network. Note that this energy function belongs to a general class of models in physics.

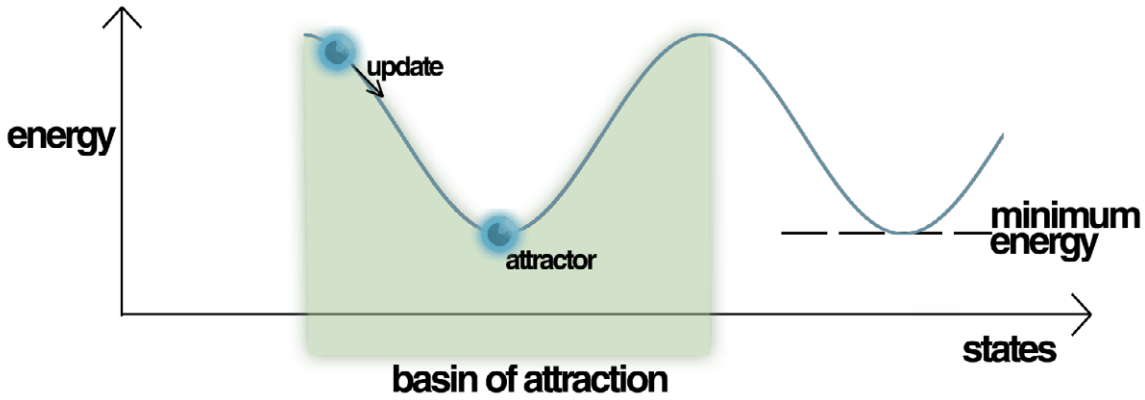



**Figure 83:** Energy Landscape of a Hopfield Network, highlighting the current state of the network (up the hill), an attractor state to which it will eventually converge, a minimum energy level and a basin of attraction shaded in green. Note how the update of the Hopfield Network is always going down in Energy.
(adapted from http://upload.wikimedia.org/wikipedia/commons/4/49/Energy_landscape.png)

### Training a Hopfield net

Initialization of the Hopfield Networks is done by setting the values of the units to the desired start pattern. Repeated updates are then performed until the network converges to an attractor pattern. In the context of Hopfield Networks, an attractor pattern is a pattern that cannot change any value within it under updating.

Training a Hopfield net involves lowering the energy of states that the net should "remember". This allows the net to serve as a content addressable memory system, that is to say, the network will converge to a "remembered" state if it is given only part of the state. The net can be used to recover from a distorted input to the trained state that is most similar to that input. This is called associative memory because it recovers memories on the basis of similarity.

There are various different learning rules that can be used to store information in the memory of the Hopfield Network. It is desirable for a learning rule to have both of the following two properties:

- *Local*: A learning rule is local if each weight is updated using information available to neurons on either side of the connection that is associated with that particular weight.
- *Incremental*: New patterns can be learned without using information from the old patterns that have been also used for training. That is, when a new pattern is used for training, the new values for the weights only depend on the old values and on the new pattern (Storkey and Valabregue 1999).

These properties are desirable, since a learning rule satisfying them is more biologically plausible. For example, since the human brain is always learning new concepts, one can reason that human learning is incremental. A learning system that would not be incremental would generally be trained only once, with a huge batch of training data.

The Hebbian Theory has been introduced by Donald Hebb (Hebb 1949), in order to explain "associative learning" in which simultaneous activation of neuron cells leads to pronounced increases in synaptic strength between those cells. It is often summarized as "Neurons that fire together, wire together. Neurons that fire out of sync, fail to link".

The Hebbian rule is both local and incremental. For the Hopfeld networks, it is implemented in the following manner, when learning $n$ binary patterns:

$$w_{ij} = \frac{1}{n}\sum_{k=1}^{n} x_i^k x_j^k$$

where $x_i^k$ represents bit $i$ from pattern $k$.

If the bits corresponding to neurons $i$ and $j$ are equal in pattern $k$, then the product $x_i^k x_j^k$ will be positive. This would, in turn, have a positive effect on the weight $w_{ij}$ and the values of $i$ and $j$ will tend to become equal. The opposite happens if the bits corresponding to neurons $i$ and $j$ are different.

The Network capacity of the Hopfield network model is determined by neuron amounts and connections within a given network. Therefore, the number of memories that are able to be stored are dependent on neurons and connections. Therefore, it is evident that many mistakes will occur if you try to store a large number of vectors. When the Hopfield model does not recall the right pattern, it is possible that an intrusion has taken place, since semantically related items tend to confuse the individual, and recollection of the wrong pattern occurs. Therefore, the Hopfield network model is shown to confuse one stored item with that of another upon retrieval.

**Human memory**

The Hopfield model accounts for associative memory through the incorporation of memory vectors. Memory vectors can be slightly used, and this would spark the retrieval of the most similar vector in the network. However, we will find out that due to this process, intrusions can occur. In associative memory for the Hopfield network, there are two types of operations: auto-association and hetero-association. The first being when a vector is associated with itself, and the latter being when two different vectors are associated in storage. Furthermore, both types of operations are possible to store within a single memory matrix, but only if that given representation matrix is not one or the other of the operations, but rather the combination (auto-associative and hetero-associative) of the two. It is important to note that Hopfield network model utilizes the same learning rule as Hebb learning rule, which basically tried to show that learning occurs as a result of the strengthening of the weights by when activity is occurring.

(Rizzuto and Kahana 2001) were able to show that the neural network model can account for repetition on recall accuracy by incorporating a probabilistic-learning algorithm. During the retrieval process, no learning occurs. As a result, the weights of the network remains fixed, showing that the model is able to switch from a learning stage to a recall stage. By adding contextual drift we are able to show the rapid forgetting that occurs in a Hopfield model during a cued-recall task. The entire network contributes to the change in the activation of any single node.

(McCullough and Pitts 1943), dynamical rule, which describes the behavior of neurons, does so in a way that shows how the activations of multiple neurons map onto the activation of a new neuron's firing rate, and how the weights of the neurons strengthen the synaptic connections between the new activated neuron (and those that activated it). Hopfield would use McCullough-Pitts's dynamical rule in order to show how retrieval is possible in the Hopfield network. However, it is important to note that Hopfield would do so in a repetitious fashion. Hopfield would use a nonlinear activation function, instead of using a linear function. This would therefore create the Hopfield dynamical rule and with this, Hopfield was able to show that with the nonlinear activation function, the dynamical rule will always modify the values of the state vector in the direction of one of the stored patterns.

# 5 Probabilistic Computation

Traditional computers often seem brilliant and simple minded at the same time. On the one hand, they can perform billions of high-precision numerical operations per second with perfect repeatability. On the other hand, they fail catastrophically when their inputs are incomplete or ambiguous. These strengths and weaknesses flow from their mathematical foundations in deductive logic and deterministic functions.

Probabilistic computation systems is working to change all this, by building the world's first natively probabilistic computers, designed from the ground up to handle ambiguity, make good guesses, and learn from their experience. Instead of logic and determinism, its hardware and software are grounded in probability distributions and stochastic simulators, generalizing the mathematics of traditional computing to the probabilistic setting. The result is a technology as suited to making judgments in the presence of uncertainty as traditional computing technology is to large-scale record keeping.

As an alternative paradigm to deterministic computation, it has been used successfully in diverse fields of computer science such as speech recognition (Jelinek 1998), natural language processing (Charniak 1993), and robotics (Thrun 2000). Its success lies in the fact that probabilistic approaches often overcome the practical limitation of deterministic approaches. A trivial example is the problem of testing whether a multivariate polynomial given by a program without branch statements is identically zero or not. It is difficult to find a practical deterministic solution, but there is a simple probabilistic solution: evaluate the polynomial on a randomly chosen input and check if the result is zero.

**Probabilistic automaton**

In mathematics and computer science, the probabilistic automaton (PA) is a generalization of the non-deterministic finite automaton; it includes the probability of a given transition into the transition function, turning it into a transition matrix or stochastic matrix. Thus, the probabilistic automaton generalizes the concept of a Markov chain or subshift of finite type. The languages recognized by probabilistic automata are called stochastic languages; these include the regular languages as a subset. The number of stochastic languages is uncountable.

The probabilistic automaton has a geometric interpretation (Rabin 1963): the state vector can be understood to be a point that lives on the face of the standard simplex, opposite to the orthogonal corner. The transition matrices form a monoid, acting on the point. This may be generalized by having the point be from some general topological space, while the transition matrices are chosen from a collection of operators acting on the topological space, thus forming a semiautomaton. When the cut-point is suitably generalized, one has a topological automaton. An example of such a generalization is the quantum finite automaton; here, the automaton state is represented by a point in complex projective space, while the transition matrices are a fixed set chosen from the unitary group. The cut-point is understood as a limit on the maximum value of the quantum angle.

# 6 Conclusion

The following Figures 84–87 demonstrate cooperation among some Soft Computing fields.

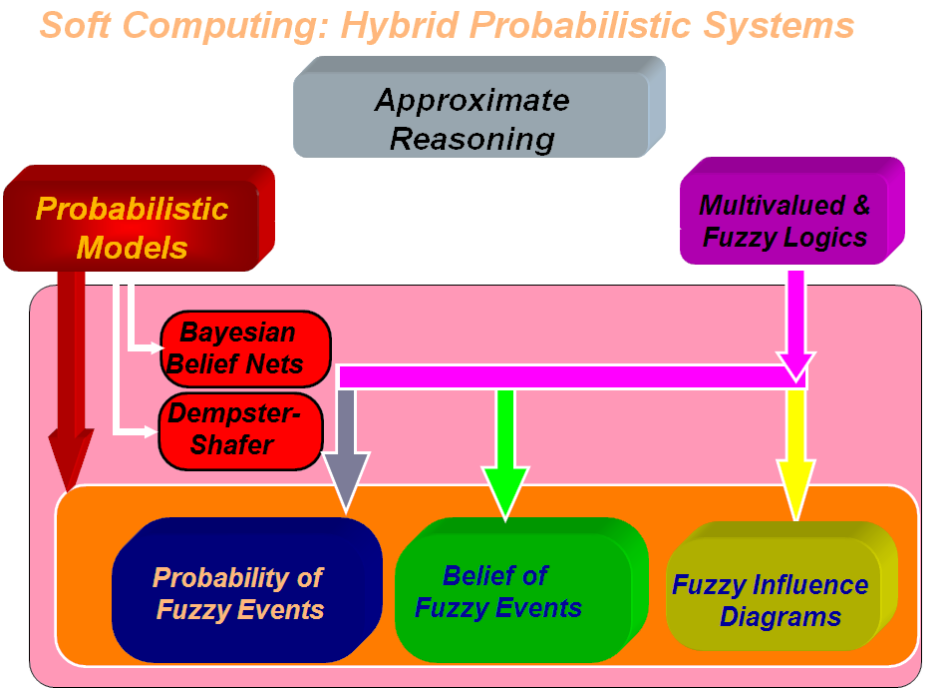



**Figure 84:** Hybrid Probabilistic Systems (adapted from http://www-bisc.cs.berkeley.edu)

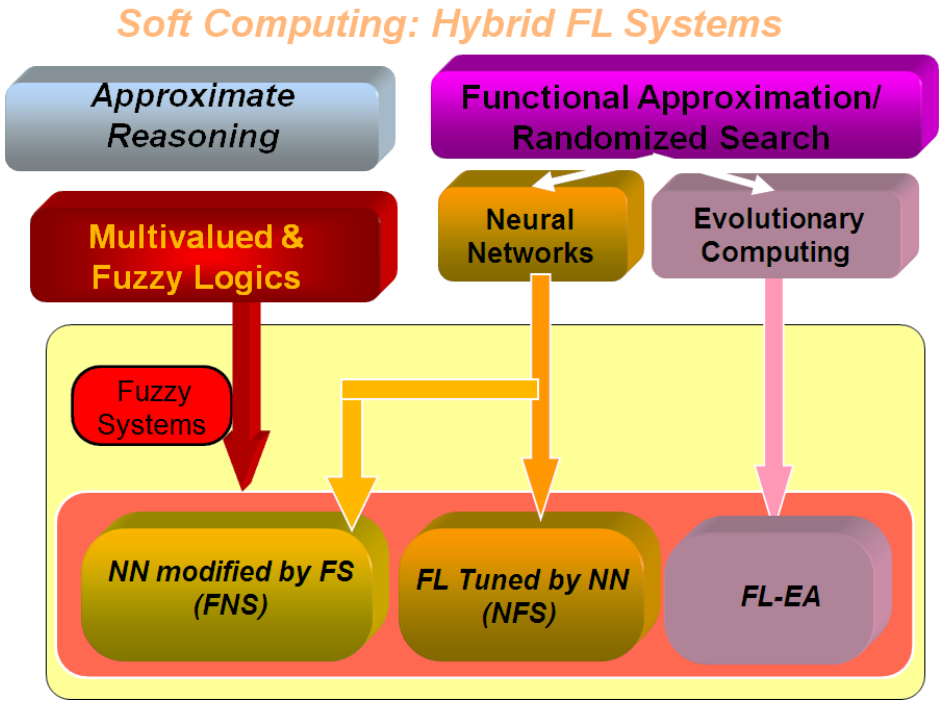



**Figure 85:** Hybrid Fuzzy Logic Systems (adapted from http://www-bisc.cs.berkeley.edu)

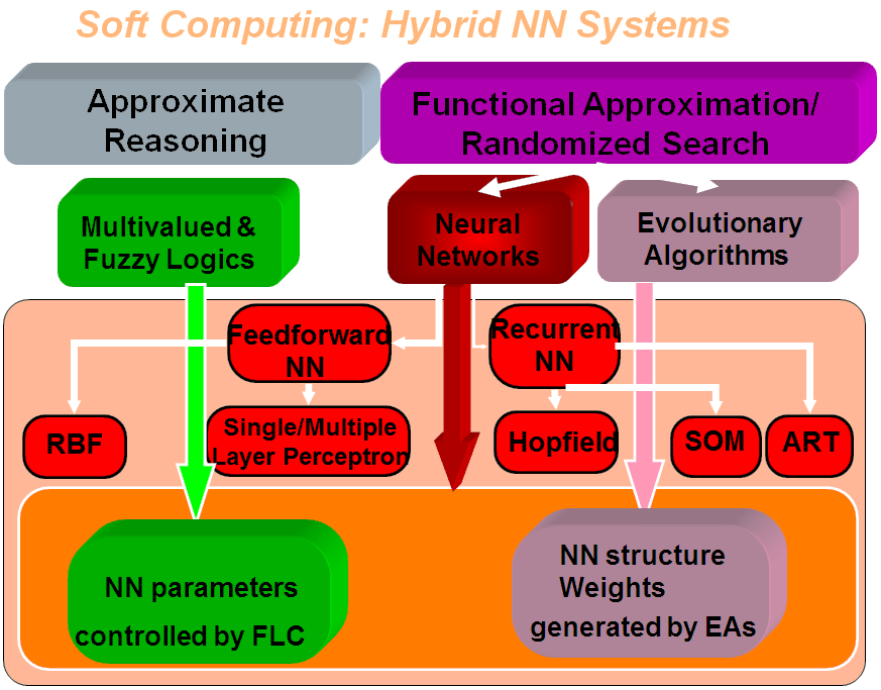



**Figure 86:** Hybrid Neural Network Systems (adapted from http://www-bisc.cs.berkeley.edu)

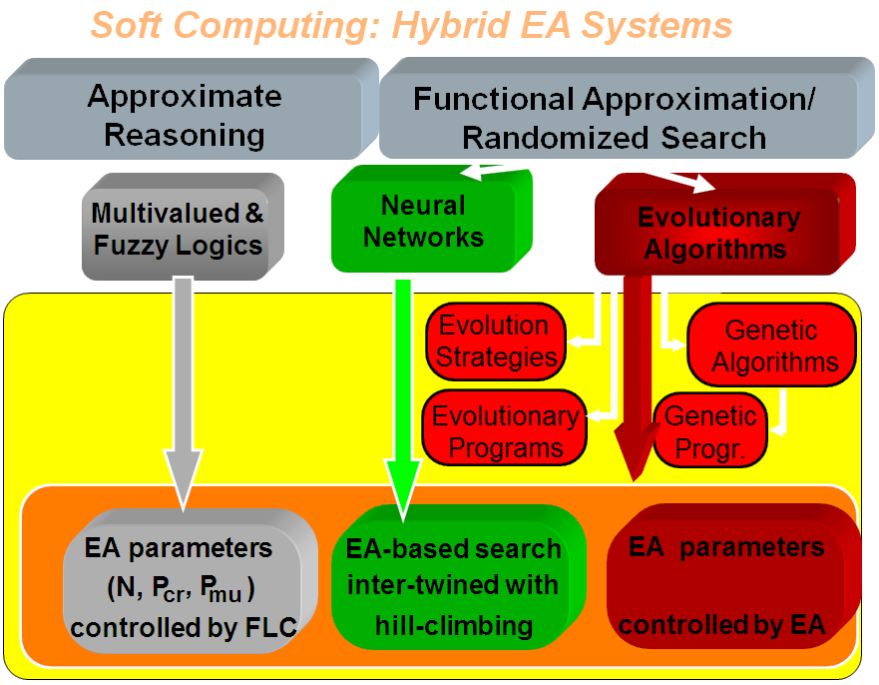



**Figure 87:** Hybrid Evolutionary Algorithms Systems (adapted from http://www-bisc.cs.berkeley.edu)

# 7 References

Bäck, T. (993): Optimal Mutation Rates in Genetic Search. In Forrest, S. (ed.): *Proceedings of the Fifth International Conference on Genetic Algorithms*, San Mateo, California, USA: Morgan Kaufmann Publishers, pp. 2–8.

Bäck, T. (1996): *Evolutionary Algorithms in Theory and Practice – Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. New York, Oxford: Oxford University Press.

Baker, J.E. (1987): Reducing Bias and Inefficiency in the Selection Algorithm. In Grefenstette, J.J. (ed.): *Proceedings of the Second International Conference on Genetic Algorithms and their Application*, Hillsdale, New Jersey, USA: Lawrence Erlbaum Associates, pp. 14–21.

Bandler, W. and Kohout, L.J. (1987): Relations, mathematical. In Sigh, M.G. (ed.), *Systems and Control Encyclopedia*. Pergamon Press, Oxford, pp. 4000–4008.

Bandler, W. and Kohout, L.J. (1988): Special properties, closures and interiors of crisp and fuzzy relations. *Fuzzy sets and Systems* 26(3): 317–332.

Beale, R. and Jackson, T. (1992): *Neural Computing: An Introduction*. J.W. Arrowsmith Ltd, Bristol, Greit Britain.

Blickle, T. and Thiele, L. (1995): A Comparison of Selection Schemes used in Genetic Algorithms. *TIK Report No*. 11, Computer Engineering and Communication Networks Lab (TIK), Swiss Federal Institute of Technology (ETH) Zürich, Switzerland.

Bonissone, P. (2002): Hybrid Soft Computing for Classification and Prediction Applications. Conferencia Invitada. In: *Proceedings of the 1st International Conference on Computing in an Imperfect World* (Soft-Ware 2002), Belfast.

Booker, L. (1987): Improving search in genetic algorithms. In Davis, L.D. (ed.): *Genetic Algorithms and Simulated Annealing. San Mateo*, California, USA: Morgan Kaufmann Publishers, pp. 61–73.

Charniak, E. (1993): *Statistical Language Learning*. MIT Press, Cambridge, Massachusetts.

Davis, L.D. (1991): *Handbook of Genetic Algorithms*. Van Nostrand Reinhold.

Fausett, L.V. (1994): *Fundamentals of Neural Networks*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

Fogel, D.B. (1994): An Introduction to Simulated Evolutionary Optimization. *IEEE Trans. on Neural Networks: Special Issue on Evolutionary Computation*, Vol. 5, No. 1, pp. 3–14.

Fogel, L.J., Owens, A.J. and Walsh, M.J. (1966): *Artificial Intelligence through Simulated Evolution*. New York: John Wiley.

Goldberg, D.E. and Deb, K. (1991): A Comparative Analysis of Selection Schemes Used in Genetic Algorithms. In Rawlins, G.J.E. (ed.): *Foundations of Genetic Algorithms*. San Mateo, California, USA: Morgan Kaufmann Publishers, pp. 69–93.

Garey, M. and Johnson, D.S. (1979): *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company.

Hebb, D. (1949): *The Organization of Behaviour*. New York, Wiley.

Hertz, J., Kogh, A. and Palmer, R.G., (1991): *Introduction to the Theory of Neural Computation*, Addison – Wesley Publishing Company. New York.

Holland, J.H. (1975): *Adaptation in natural and artificial systems*. MIT Press.

Jelinek F. (1998): *Statistical Methods for Speech Recognition (Language, Speech, and Communication)*. MIT Press, Boston, MA.

Klir, G. and Yuan B. (1995): *Fuzzy Sets and Fuzzy Logic: Theory: and Applications*. Prentice Hall, NJ, USA.

Kohout, L. (1999): *Notes on Fuzzy Logics. Class Notes, Fuzzy Systems and Soft Computing*. Department of Computer Science, Florida State University, Tallahassee, FL, USA.

Kohout, L. (2000): Foundations of Knowledge Engineering for Systems with Distributed Intelligence: A relational Approach. *Encyclopedia of Microcomputers*. Kent, A. and Williams, J. (eds.), Vol. 24, Supplement 3, Marcel Dekker Inc., NY, USA.

Koza, J.R. (1992): *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA: The MIT Press.

Li, X., Ruan, D. and van der Wal, A.J. (1998): Discussion on soft computing at FLINS'96. *International Journal of Intelligent Systems*, 13, 2-3, 287–300.

McCullough, W.S., and Pitts, W.H. (1943): A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5, 115–133

Michalewicz, Z. (1994): *Genetic Algorithms + Data Structures = Evolution Programs*, Second, Extended Edition. Berlin, Heidelberg, New York: Springer-Verlag.

Moussa, A.d and Kohout L. (2001): Using BK-Products of Fuzzy Relations in Quality of Service Adaptive Communication. In: *Proceedings of IFSA/NAFIPS-2001*, pp. 681–686, IEEE, Vancouver, Canada.

Mühlenbein, H. and Schlierkamp-Voosen, D. (1993): Predictive Models for the Breeder Genetic Algorithm: I. *Continuous Parameter Optimization. Evolutionary Computation*, 1 (1), pp. 25–49.

Mühlenbein, H. (1994): The Breeder Genetic Algorithm – a provable optimal search algorithm and its application. *Colloquium on Applications of Genetic Algorithms*, IEE 94/067, London.

Pohlheim, H. (2006): *Evolutionary Algorithms: Overview, Methods and Operators*. Available from www.geatbx.com.

Price K. (1999): An Introduction to Differential Evolution, In Corne, D., Dorigo, M. and Glover, F. (Eds.) *New Ideas in Optimization*, McGraw-Hill, pp. 79–108.

Rabin, M.O. (1963): Probabilistic Automata, *Information and Control 6* pp. 230–245.

Rechenberg, I. (1973): *Evolutionsstrategie – Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Stuttgart: Frommann-Holzboog.

Rizzuto, D.S. and Kahana, M.J. (2001): An autoassociative neural network model of paired-associate learning. *Neural Computation*, 13, 2075–2092.

Russell I., Markov Z., Holder L., eds. (2005), Machine Learning and Neural Network Approaches to Feature Selection and Extraction for Classification, *Special Issue of the International Journal of Pattern Recognition and Artificial Intelligence*.

Schwefel, H.-P. (1981): *Numerical optimization of computer models*. Chichester: Wiley & Sons.

Spears, W.M. and De Jong, K.A. (1991): An Analysis of Multi-Point Crossover. In Rawlins, G.J.E. (ed.)*: Foundations of Genetic Algorithms*. San Mateo, California, USA: Morgan Kaufmann Publishers, pp. 301–315.

Storkey, A.J. and Valabregue, R. (1999): The basins of attraction of a new Hopfield learning rule. *Neural Networks*, pp. 869–876.

Storn, R. and Price, K. (1997): Differential Evolution – A simple and efficient adaptive scheme for global optimization over continuous spaces. *Global Optimiz.*, vol. 11, pp. 341–359

Syswerda, G. (1989): Uniform crossover in genetic algorithms. In Schaffer, J.D. (ed.): *Proceedings of the Third International Conference on Genetic Algorithms*, San Mateo, California, USA: Morgan Kaufmann Publishers, pp. 2–9.

Thede, S.M. (2004): *An introduction to genetic algorithms, Fundamentals*, In JCSC 20, pp. 115–123.

Thrun, S. (2000): Probabilistic algorithms in robotics. *AI Magazine*, 21(4):93–109.

Verdegay, J.L., Ed. (2003): Fuzzy Sets-based Heuristics for Optimization. *Studies in Fuzziness*. Springer Verlag.

Zadeh, L.A. (1994): Soft Computing and Fuzzy Logic. *IEEE Software* 11, 6, 48–56.

Zadeh, L.A. (2001): Applied Soft Computing. *Applied Soft Computing* 1, 1–2

# 8 List of Figures

# 9 List of Tables

# 10 Endnotes

1. Adapted from http://www.myreaders.info